SIX-MONTH REPORT

APRIL 1 to SEPTEMBER 31, 1994

INVESTIGATION OF MODULARLY CONFIGURED ATTACHED PROCESSORS WITH INTELLIGENT MEMORIES

GRANT NO. N00014-93-1-1343

1. STATUS OF PROJECT

Objective 1 (register-level design of MCAP): The register-level design of the MCAP has been completed for all component types except the instruction component, which has been partially designed. The component designs are currently being documented by Glenn Gibson and an undergraduate assistant. This design effort has resulted in a masters-level thesis.

Objective 2 (architecture/algorithm case studies): This study has progressed slowly because of the incompleteness of the simulator. However, the simulator has just been completed and considerable progress on this objective is expected in the next six months. Preliminary work has been done by assuming all data needed by an algorithm is already in the MCAP. Also, Sergio Cabrera and his post-doctoral assistant have investigated which algorithms should be studied in the area of signal and image processing and Yi-Chieh Chang and a masters-level student have simulated some matrix operations. An undergraduate assistant and a doctoral student have just started work in this area and a masters-level student is to be added in the near future. Chang, Gibson and a masters-level student have produced a conference paper on matching matrix multiplication to an MCAP architecture.

Objective 3 (two memory controller designs): Yu-Cheng Liu and a masters-level student began their work on comparing two memory controller designs in August when our new workstations arrived and the Mentor Graphics design software was installed on them. This work will continue throughout the coming year. A related study by Gibson, Liu and Chang of memory hierarchies and computational intensity has resulted in a conference paper.

Objective 4 (technology evaluations): Considerable work has been done in this area. This work has mainly been concerned with implementing an MCAP on a multichip module and has been carried out by Vijay Singh and two masters-level students with some assistance by Gibson. Some investigation of a wafer-scale implementation has been done by Chang. This work has produced a master's thesis and two conference papers. Another paper has been submitted to a journal.

Objective 5 (simulator development): Except for updating the architecture editor, the simulator software package has been completed. However, because it is difficult to program an MCAP using the current assembler, a preprocessor is being programmed by Chang and an undergraduate assistant. Gibson and an undergraduate assistant will continue testing and documenting this software. This work has produced two theses and a third is currently being written. A conference paper is being written and will be submitted before October 15. An expanded version of this paper will be submitted to a journal.

2. MASTERS-LEVEL THESES PRODUCED

Ernesto Castro-Gomez, "Assembler Design and Algorithm Implementation on a Modularly Configured Attached Processor," Thesis, University of Texas at El Paso, 1994.

Alejandro Brito, "A Graphics Editor for the MCAP Simulator," Thesis, University of Texas at El Paso, 1994.

Sanjay Singh, "A Comparative Evaluation of Implementing a Novel Modularly Configured Attached Processor Architecture," Thesis, University of Texas at El Paso, 1994.

Michael Flahie, "Fast N-bit Multipliers Using Cascaded Half Adders," Thesis, University of Texas at El Paso, 1994.

Stephen Synesyzn, "Simulation of a Memory Controller for a Modularly Configured Attached Processor," Thesis, University of Texas at El Paso, currently being written.

3. PAPERS ACCEPTED FOR PUBLICATION (see attachments)

G. A. Gibson, Vijay Singh, Sanjay Singh, Y. C. Liu, Y. C. Chang and Sergio Cabrera, "MCM Implementation of Modularly Configured Attached Processors," *IEEE International Computer Symposium,* Taiwan, Dec., 1994.

G. A. Gibson, Y. C. Liu and Y. C. Chang, "Application of Computational Intensity to Memory Hierarchy Design," *IEEE International Computer Symposium,* Taiwan, Dec., 1994.

Y. C. Chang, G. A. Gibson and Claudia Ayala, "Simulation and Performance Evaluation of a Modularly Configurable Attached Processor," *IEEE International Conference on Parallel and Distributed Systems,* Taiwan, Dec., 1994.

Y. C. Chang, J. H. Kim and Saji Jorge, "A Module-sliced High-yield WSI Memory System," *IEEE International Conference on Wafer Scale Integration,* USA, Jan. 1995.

4. PAPERS SUBMITTED FOR PUBLICATION

Sanjay Singh, B.W. Gremel, Vijay Singh and G. A. Gibson, "Design Issues in a CMOS Implementation of a Modularly Configured Attached Processor," submitted to *Int'l J. of Electronics*.

S. W. Wu, Alejandro Brito and Sergio Cabrera, "Design of a Modularly Configured Attached Processor for FFT Computation," abstract submitted to *ICASSP-95*.

ATTACHMENT A

MCM IMPLEMENTATION OF MODULARLY CONFIGURABLE ATTACHED PROCESSORS

Glenn Gibson, Vijay Singh, Sanjay Singh,
Yu-cheng Liu, Yi-Chieh Chang, and Sergio Cabrera

Department of Electrical and Computer Engineering
The University of Texas at El Paso
El Paso, Texas 79968-0523

Abstract

A new architecture for high-performance parallel attached processors is described in this paper. Based on this architecture, an attached processor can be implemented as multiple memory-to-memory pipelines, each being constructed with a class of fundamental components. The unique features are that the attached processor can be configured to match a set of algorithms and its memory controllers can be programmed to fit the access patterns required by the algorithms. As a result, high utilization of the processing logic for given sets of algorithms can be obtained. An example based on matrix multiplication is used for illustration. Finally, design issues related to the implementation of the attached processor based on an MCM technology are discussed. [1]

1 Introduction

An attached, or back-end, processor is a processing system that is connected to a host computer for the purpose of very quickly executing most of the overall system's computational tasks. Typical early attached processors were the AP-120B and FPS-164 made by Floating Point Systems, Inc., the IBM 3838, and the MATP made by Datawest, Inc. [1], [2], [3]. These attached processors all have their own data memories and transfer data between these memories and the main memories of their hosts using DMA data channels. They also include their own code memories where subprograms may be permanently stored or downloaded from their hosts. These subprograms are initiated by commands from the host and supervise the data flows from the attached processor's data memories, through the attached processor's processing elements, and back into the data memories.

Although the early attached processors included limited multiprocessing, the more recently implemented processing arrays are also controlled by a host (e.g., the PAX computer [4]) and are designed to perform most of the overall system's computational tasks. Therefore, these arrays and even the array processing portions of today's supercomputers, such as the Cray series [1], [3] could be interpreted as attached processors.

The specific purpose of an attached processor is to execute members of a set of algorithms very quickly. The broader the set of algorithms the more generally applicable the attached processor. The underlying goal of the designer is to efficiently utilize the hardware for as broad a set of algorithms as possible. However, for most current designs, the average sustainable execution rates have been found to be only 5% to 20% of their peak rates, which are determined by summing the maximum computational rates of the processing elements. For example, the sustainable rate for a Cray X-MP with four processors may be as low as 5% for some algorithms [5]. Also extensive evaluations of recent high-performance computations using Lapack are given in [6] and using NSA parallel benchmarks are given in [7]. Although some of the lost efficiency is necessitated by the algorithms, much of it is due to memory accessing and contention for shared resources in general, including internal buses.

Described in this paper is a class of high-performance attached processors called Modularly Configurable Attached Processors (MCAPs) which can attain quickness and high utilization through: (1) Closely matching their architectures to the set of algorithms they are to execute. (2) Overlapping of processing and memory accessing by using memory prefetching. (3) Minimizing the movement of data. (4) Using a high-speed technology with MCM or wafer scale implementations.

An MCAP is constructed from the component types specified in Sec. 2. These component types are such that each member of the class may include parallel processing, memory-to-memory pipelines, and be constructed in a building block fashion. They encompass routing components (including buses) as well as memory, control, and processing components. By overlapping processing with

[1] The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency.

memory accessing and matching an architecture with a set of algorithms, it is predicted that the average sustainable rate for a specific set of algorithms can attain at least 60% of the peak rate. By defining components that are simple enough to be fabricated onto single low-density ICs, a high-speed technology may be used.

Much of an MCAP's efficiency is gained by distributing the instructions for the next algorithm (or algorithm phase) to the various components while the current algorithm (or phase) is executing. Once the algorithm begins, these instructions dictate the modes, routing patterns, prefetching patterns, and so on of the components receiving them. After an algorithm starts, each component operates more or less on its own except for responding to its handshaking signals. Efficiency is further enhanced by prefetching operands from the memory subsystems. Prefetching using programmed patterns avoids the misses that result from using ordinary caches.

Section 2 describes the architecture of the MCAP and the fundamental components required to construct an MCAP. Section 3 illustrates how to match an algorithm with a given MCAP architecture in order to attain a high sustainable rates. A major issue related to the implementation of MCAPs is the choice of semiconductor technology and packaging, which affect speed, gate density, power dissipation, and cost. The emphasis of implementation considerations given in this paper is on CMOS Multi-Chip Module (MCM) technology due to its ability to achieve fast inter-chip communication. Section 4 discusses various design issues involved using the MCM approach to implement the MCAPs. Such considerations include transistor count, loading, estimate of speed, and power dissipation.

2 MCAP Architecture

An MCAP is an attached processor that is constructed entirely from a standard set of connections and components. This standard set consists of two types of asynchronous connections and twelve types of components. The definitions of the connection and component types provide a standard set of rules that allow the components to be easily configured in different ways to construct attached processors that can efficiently perform different sets of algorithms.

An MCAP has exactly one instruction component and it is connected to a memory component for storing instructions. Most of this memory component is a ROM that contains the subprograms needed to execute the algorithms, but some of it is a RAM that can receive instructions (those that initiate the subprograms) from the host.

An MCAP operates by drawing an instruction stream from the memory component into the instruction component. The instruction component uses internal instructions in the stream to form external instructions that are then distributed to the other non-memory components through the MCAP's (one and only) bus component. The instruction stream is illustrated in Fig. 1. Note that all components in the instruction stream include input instruction queues. When the non-memory components have received all of the instructions needed to perform an algorithm, they automatically prefetch the data from the memory components, route the data to and from the processor components and store the results back into the memory components. Some controller components, which are the components that supervise all memory accessing, are used to automatically transfer data between the host's main memory and the MCAP's memory components. The instruction and data streams are separate, thereby allowing the instructions needed for the next algorithm to be distributed while the current algorithm is executing.

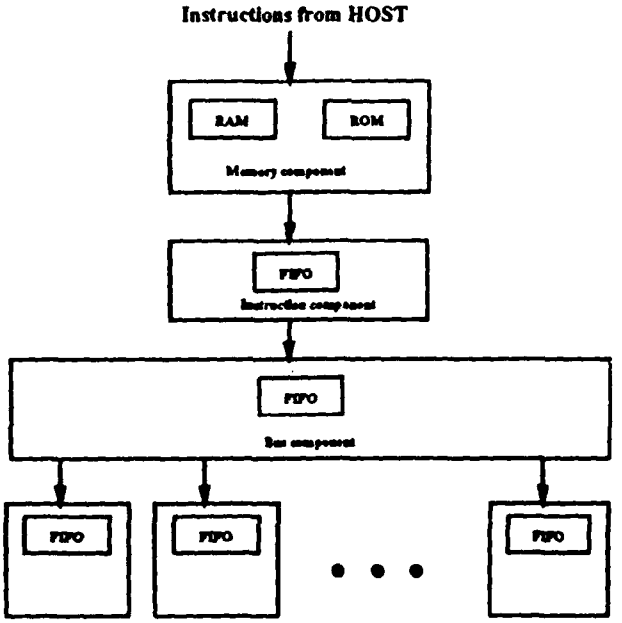

Fig. 1. The instruction stream

The two types of connections are referred to as instruction and data connections. These connections are asynchronous and, therefore, must include handshaking lines as well as data and, perhaps, address lines. Because memory components are connected to controller components only, they are an integrate part of controller/memory subsystems. Therefore, the exact controller/memory connection specificatios are left to the subsystem designer and may be synchronous.

Instruction connections are for passing instructions from the instruction component to the bus component and from the bus component to one of the other non-memory components. An instruction connection consists of unidirectional instruction and address buses and a Req/Ack handshaking pair. The component that is to receive the instruction is indicated by the a component number on the address bus. A transfer is initiated when the sending component puts

an address on the address bus, an instruction on the instruction bus and begins the handshaking. Except for the connections to memory components, all connections used to transfer data are data connections. They are used to pass data to and from the processors and consist of only a unidirectional data bus and a Req/Ack pair. A data transfer consists of placing data on the data bus and initiating the handshaking. Except for a write to a memory component, all transfers include the latching of an instruction or datum into a queue at the receiving end.

The twelve types of components are divided into six categories as indicated below:

- Instruction
- Bus
- Memory
- Processor
  - Elementary–one input, one output
  - Two-input–two inputs, one output
  - Comparator–two inputs, one output plus special outputs
- Router
  - Join–multiple inputs, one output
  - Fork–one input, multiple outputs
  - Link–multiple inputs, multiple outputs
- Controller
  - RAM–internal to MCAP, no partitions
  - Single-access–internal to MCAP, has partitions
  - Dual-access–connects to main memory, has partitions

As mentioned earlier, an MCAP contains one memory component for storing instructions, one instruction component for executing internal instructions and forming external instructions, and one bus component for distributing the instructions. An MCAP may contain several controller, router, and processor components and several other memory components for storing data. However, the other memory components can be connected to controller components only. Only controller components are capable of being programmed to prefetch data from and deposit data into data memory components. Although the instruction memory component or a dual-access component can be connected to the host system, all other components can be connected to the MCAP's components only.

Each non-memory component that is used during the execution of an algorithm contains an instruction input queue, one or more data input queues, and control logic that includes a number of registers. For example, a typical elementary component is shown in Fig. 2. The instructions for an algorithm received by a component fill these registers and then the register contents dictate the activity within the component while the algorithm is executed. They determine the component's mode and, for a routing component, the patterns for accepting inputs and distributing outputs. For a controller component, they determine the memory partitions, DMA accessing patterns, and patterns for prefetching the operands needed by the algorithm.

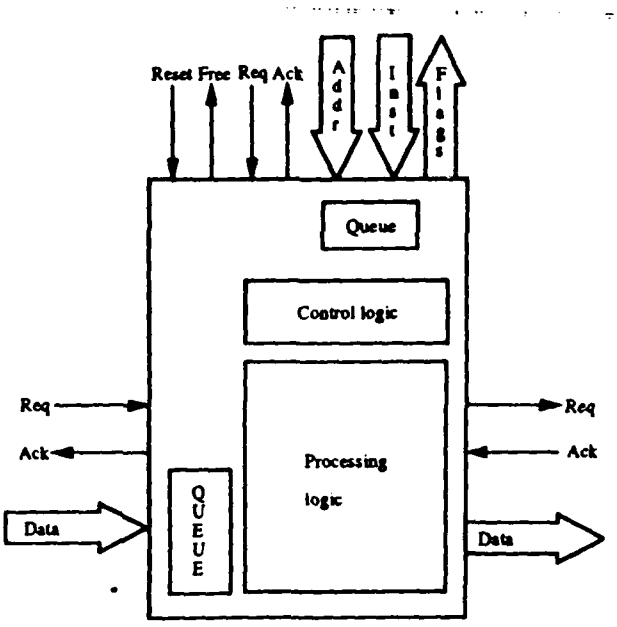

Fig. 2. Block diagram of an elementary component

Each of the components that receives instructions contains a Number of Operands Output (NumOpsOut) register that is always the last register filled before the component begins its part in the execution of the algorithm. Each time the component outputs an operand, the NumOpsOut register is decremented. When the NumOpsOut register becomes zero, the component has completed its part in executing the current algorithm. It may then distribute new values, those needed for the next algorithm, from its instruction input queue to its registers. This cycle may continue indefinitely. Except for reacting to the handshaking (i.e., Req and Ack) signals in its connections, each component acts independently. The data is input to a data queue through an input connection, processed or routed through a bus, and output through an output connection. Because separate queues are used to input instructions and data, the instruction and data streams are completely separate.

The processor components are used for performing unary and binary arithmetic/logic operations. There are three types of processor components. There are one-input elementary (E) components, two-input (T) components, and comparator (C) components. These components contain only two registers, a mode register and a NumOpsOut register. The mode register dictates the actions taken by the component and the NumOpsOut register gives the total number of operands that is to be output before the current algorithm is completed. Both the E and T components may be used for either unary or binary operations, depending on the mode. When an E component is used for

a binary operation it must, of course, input both operands through its single input connection. A T component performing a unary operation would use only one of its two input connections.

A C component is similar to a T component, but has two special sets of lines connecting it to the instruction component. There can be only one C component in an MCAP. As usual, its current function is determined by its mode. One of its functions is to simply compare two inputs and set relational flags that are then transmitted to the instruction component over one set of the special lines. When performing comparisons, there are no outputs other than the flag outputs. The C component can, however, also determine the maximum or minimum of a sequence of numbers. In this case, the second set of special lines is used to output the index of the maximum or minimum to the instruction component. The maximum or minimum is output on the output data connection.

Routing components are for directing data along the proper paths. There are three types of routing components, join (J) components with more than one input and one output, fork (F) components with one input and more than one output, and link (L) components with more than one input and more than one output. In addition to the mode and NumOpsOut registers, they contain registers for dictating their input and output patterns while the current algorithm is being executed. F and L components may include broadcasting in their output patterns. J and F components may be used in conjunction with T and E components to form pipelines with feedback that can accumulate sums.

There are three types of controller components, RAM (R) components, single-access (S) components, and dual-access (D) components. All controller components are for automatically retrieving operands from and storing results in their associated memory components. In addition, a D component includes connections communicating with the host's main memory. All controller components have an output data connection for outputting operands to the remainder of the MCAP and an input data connection for inputting results from the MCAP. Therefore, they must be capable of handling both an output data stream and an input data stream. A queue is inserted in each of these data streams. A D controller also has input and output data stream for transferring data to and from the host's main memory.

A significant difference between the controller components and the other programmable components is that a Number of Operands In (NumOpsIn) register as well as a NumOpsOut register must be included. The NumOpsIn register serves the same purpose for the input data stream as NumOpsOut does for the output stream. An S component differs from an R component in that its memory may be divided into partitions that consist of blocks of memory having consecutive addresses. The memory components are interleaved so that the partitions, because they occupy consecutive addresses, are spread across the components.

In addition to the mode, NumOpsOut and NumOpsIn registers, an S component contains registers for specifying the patterns for accessing the partitions and a set of registers for each partition for specifying the pattern of accesses within the partition.

That portion of a D component that communicates with the MCAP is similar to an S component except that a NumOpsOut register is needed for each output stream and a NumOpsIn register is needed for each input stream. Both S and D components include a window which is a set of memory locations with consecutive addresses whose base address increments after each repetition of a pattern. The purpose of the window is to separate the input from the output. Data that are output from the partition must involve accesses that are within the window and inputs to the partition must involve accesses that are outside the window. Because a partition is treated as a circular memory, the location with the highest address in the partition is considered to be adjacent to the one with the lowest address and the window is considered to move in a circle.

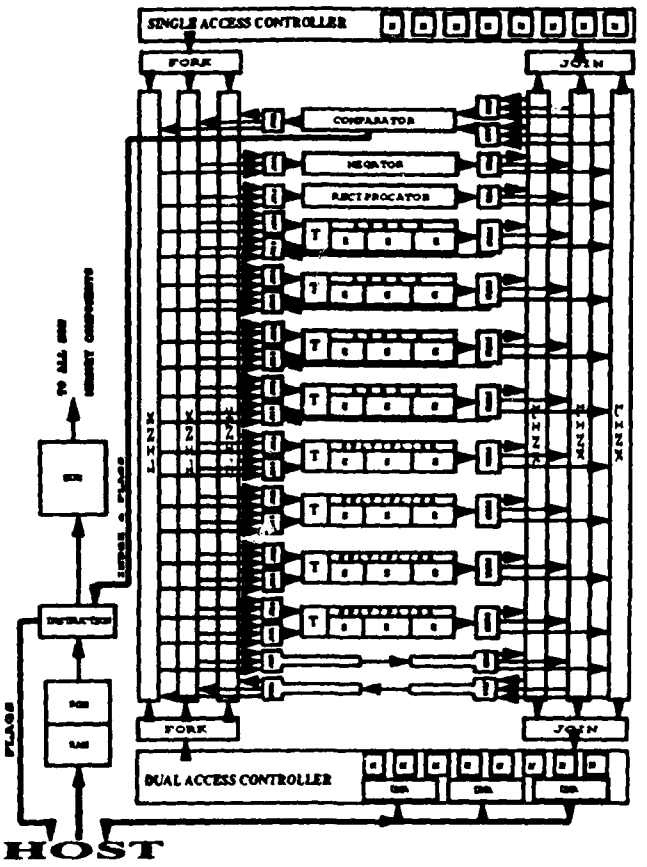

Fig. 3. An example MCAP architecture

An example architecture is given in Fig. 3. Its processing subsection includes a comparator (C component), a negator (E component), a reciprocator (E component), a set of four pipelined adders capable of accumulation, and a set of four pipelined multipliers. Each adder or multiplier is con-

structed of four stages (a T component followed by three E components). All communications to and from the processing components are through six L components, three on each side of the processor. J and F components are provided to allow flexible use of the L components. Also, to allow for accumulation there is a feedback connection between the F component at the output from each adder and the J component at the input to the adder. There is a D component to provide intermediate memory and a connection to main memory. The S component provides internal storage.

3 Matching Algorithms to Architectures

In order to efficiently use the available logic and interconnections, an architecture must be carefully matched to an algorithm or set of algorithms. This involves a study relating the flows, storage and processing of the data required by the algorithm(s). Clearly, there is no point in increasing the speed of a processing subsystem if the current interconnections and memory hierarchy are inadequate to support the processing (or vice versa). But a good balance for one algorithm may not be a good balance for a different algorithm. What is needed is a satisfactory tradeoff for the work mix expected of a system and a means of evaluating the design parameters chosen.

Space allows only a single example, so let us consider the computation that most frequently occurs in computationally intense algorithms, matrix multiplication. Let us examine how the MCAP in Fig. 3 could be analyzed relative to the algorithm $AB = C$ using the middle product method [3] where A, B and C are $n \times n$ matrices. Fig. 4 shows the required flow of data through the MCAP. The variable m is the number of rows that can be simultaneously stored in each of the D and S component memories. The expressions give the total numbers of operands transferred between the major subsystems.

The algorithm consists of the computations

$$\sum_{i=1}^{n} a_{ij} B_j = C_j \qquad i = 1, \ldots, n$$

where the a_{ij}s are the elements of A, the B_js are the rows of B, and the C_is are the rows of C. The algorithm proceeds by storing the first m elements of the first column of A and the first m rows of B in the D component's memory. Then the products $a_{i1} B_i$, for $i = 1, \ldots, m$, are formed and stored in the S component. Next, the first m elements of the second column of A are brought into the D component and the products $a_{i2} B_i$ are formed and added to the corresponding previous products, with the results being returned to the S component. This is repeated $n/m - 1$ times, but the last time the product totals, which are the first m rows of C, are put in the D component and then output to main memory. The entire process is repeated n/m times. Overlapping can be used to reduce the required time.

By matching this algorithm with the architecture in Fig. 3, it is seen that each adder and multiplier must perform approximately $n^3/2$ operations and each link on the left and two of the links on the right must perform approximately n^3 transfers. (The third link on the right is not be needed.) The approximate numbers of accesses to the S component, D component and main memory are about $2n^3$, $n^3(1 + 1/m)$ and n^3/m, respectively. If T is the per stage processing time of the multipliers, then T should also be the per stage processing time of the adders and $T/4$ should be the transfer time of the links. The access times of the S component, D component and main memory should be $T/8$, $mT/4(m + 1)$ and $mT/4$, respectively for both reads and writes. For T = 40 ns and m = 8, the link transfer time should be 10ns and the average memory access times should be 5 ns, 9 ns and 80 ns. The computation rate would be 200 Mflops per second. If the MCAP were put into an MCM or wafer and memory interleaving were used, these times would certainly be within the capability of current HCMOS technology. (The join and fork components were ignored in this discussion because the communication times are dictated by the slower link components.) BiCMOS and GaAs could produce proportionately faster processing, memory and memory controller components, but, as seen in the next section, increasing the speed of the link components is a more challenging problem.

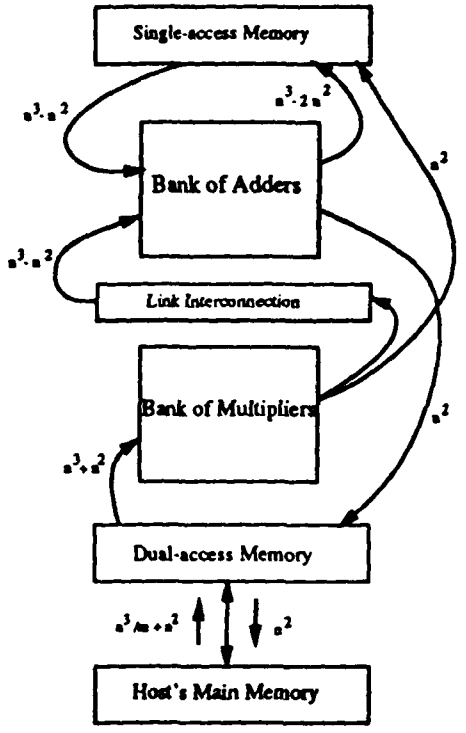

Fig. 4. Data flow for matrix multiplication

Except for the unused link component, the design would utilize the link and processor components over 95% of the time while performing a matrix multiplication. In contrast,

note that matrix addition would utilize these components only about 50% of the time on the average with the S, multiplier and some of the routing components not being used at all. This contrast points out the need for different designs for different algorithms and the need for compromise when a set of algorithms must be executed on the same architecture.

4 MCM IMPLEMENTATION CONSIDERATIONS

Since the signal delays associated with a PCB implementation are expected to be prohibitively excessive, it is thought that the fabrication of an MCAP in a Multi-Chip Module (MCM) configuration or Wafer Scale Integration (WSI) are the only realistic alternatives for attaining high-performance. Some important design considerations for implementing an example MCAP architecture in MCM configuration are presented in this section.

Fig. 5 shows a layout for an MCM implementation of the example architecture. In designing this layout, we aimed toward minimizing chip to chip interconnections, maximizing interconnection densities, and using a parallel architecture. Other factors of importance are ground and power plane generation and physical design verification. The amount of heat generated is directly dependent on the type of substrate (MCM's are classified according to the substrate technology; MCM-C, MCM-D, and MCM-L), selection of bonding and placement of chips. Parasitics on the interconnects, inductances on the power lines and the I/O pin limitation are other important considerations.

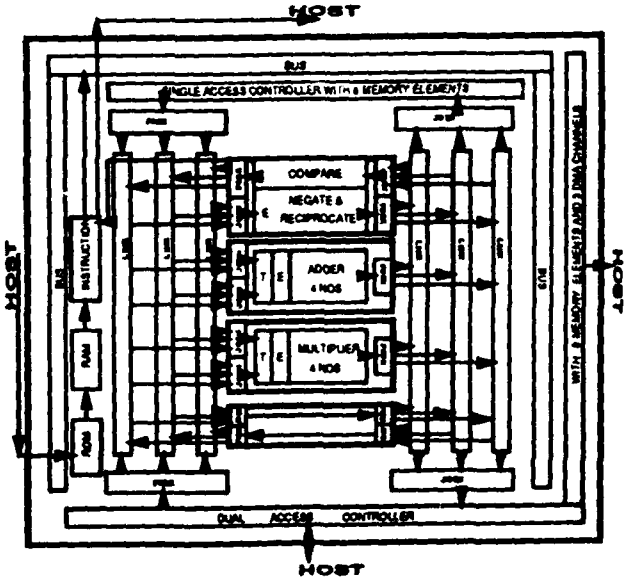

Fig. 5. Layout of MCAP on an MCM

4.1 The transistor count

In estimating the total number of transistors required to build the proposed MCAP, we made the assumption that the technology used is high-speed CMOS. CMOS was picked as the first benchmark technology because of its commercial maturity. In the future, faster technologies such as GaAs will be evaluated. As an example, let us consider a pipelined 64-bit floating point adder with four stages. It has: (1) nine 64-bit registers with 4032 transistors (7 transistors per bit for a dynamic latch), (2) seventy-four 2-input XOR gates with 592 transistors, (3) one hundred and twenty-six 2 to 1 MUX's with 504 transistors, (4) two 11-bit adders with 528 transistors, (5) one 52-bit adder with 1248 transistors, (6) a 64-bit leading zero detector with 5000 transistors, (7) two 52-bit barrel shifters with 4000 transistors, and (8) rounding and other control logic taking 6500 transistors.

The total is 23K transistors for an adder. By having four pipelined stages, we can achieve stage delays of less than 20 ns [8]. This delay is of course expected to be even smaller for faster technologies like GaAs. Similarly, we can evaluate the number of transistors for a pipelined 64-bit floating point multiplier (using an optimized, modified Booth's algorithm) and arrive at a total of 58K transistors. Again with four pipelined stages, the delay per stage is less than 20 ns [8]. Following this procedure, the transistor count for the rest of the elements in the MCAP are calculated and Table 1 gives the count for the various components. A figure of approximately ten million is reckoned as the transistor count to build the whole MCAP.

In the proposed architecture, the bottleneck is the communication through the LINK elements because of their high fan-out and relatively large interconnection distances. This means that the output buffers for these elements must be relatively large. Next, we present the delay, power and area calculations for the output buffers as functions of *fan-out* (F) and *interconnection length* (ℓ). For these calculations, (1) The input capacitance of a gate including the lead and ESD capacitance is $C_{in} = 1$ pF; (2) The width of the metal conductor used for an interconnection is $w = 25\mu$m; (3) The capacitance of the metal conductor is $C_{m1} = 30$ aF/μm^2; (4) The sheet resistance of the metal is $R_s = 0.05\Omega/\Box$; (5) The feature size is $\lambda = 0.5\mu$m.

4.2 Load capacitance

For the load capacitance $C_l = C_{int} + F \times C_{in}$, with $C_{int} = w \times \ell \times C_{m1} = (0.025) \times \ell \times 30 \times 10^{-18} \times 10^6$ pF $= 0.75 \times \ell$ pF where ℓ is in mm and $C_{in} = 1$ pF. The resistance of the interconnect is

$$R_{int} = R_s \times (\ell/w) = 0.05 \times (\ell/0.025) = 2 \times \ell \ \Omega \quad (1)$$

Thus, a LINK element with a fan-out of 19 and with an average interconnection length of 2 cm has load capacitance of 34 pF.

4.3 Average delay

It is known that, in general, the minimum size of a logic gate has a W/L ratio of 2. So, we start with a ratio of

2 and go to higher values in stages in order to drive a load within a short time. By dividing the buffering stages into the number of buffers with increasing W/L, optimum speeds can be achieved. It has been found that a stage ratio of 3 [9] gives best results. Also, the optimum number of stages is $N = 0.91(\ln C_l + 4.19)$, where N is truncated to the nearest integer. Using the optimum number of stages, the average delay is

$$T_{avg} = 0.484(N-1) + 5C_l/3(N-1) + 0.076 \text{ ns} \quad (2)$$

The plot of T_{avg} as a function of F and ℓ is shown in Fig. 6. For the example with F = 19 and ℓ = 20 mm, the delay time is seen to be 3.2 ns.

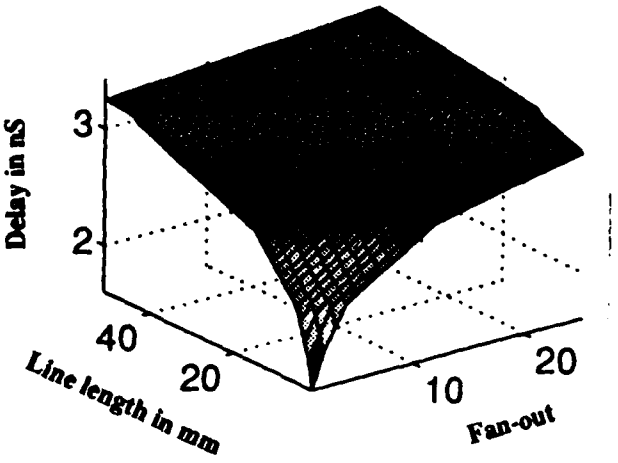

Fig. 6. Buffer and interconnect delay

4.4 Buffer area

A simple inverter with $(W/L)_n = (W/L)_p = 2$ will need an area of 66 μm^2. A buffer with equal rise (t_r) and fall (t_f) times requires $(W/L)_p = 2(W/L)_n = 4$ and the area is going to be 150 μm^2. The total area of the buffer depends on the number of stages and, hence, is a function of F and ℓ. We have Area $= 66 + 3[14(N-1) + 36(1 + 3 + 3^2 + \cdots + 3^{N-2})] \approx 55 \times 3^{N-1}$ μm^2

4.5 Power dissipation in the buffer

In CMOS, most of the power is dissipated during switching and, hence, dynamic power is approximately equal to the total power. The dynamic power is $P_d = C_T \times v^2 \times f_{avg} = 25(C_l + C_{buff})/T_{avg}$, where $C_{buff} = 0.0152(3^{N-1})$ pF

Since the design of an MCAP uses asynchronous communication, the transfers over a LINK component involves the return of an acknowledge signal and the transmission of an output enable signal. It is estimated that the transfer rate may be as high as $f = 1/2T_{avg}$ Hz. For $F = 19$ and $\ell = 20$ mm, the total power dissipated by buffer is 175 mW.

4.6 Thermal management

There have been successive revolutions in device technologies, proceeding from TTL, ECL and NMOS to the recent high-speed CMOS, BiCMOS and GaAs. Three to five orders of magnitude reduction in minimal feature size, an order of magnitude in the characteristic chip dimension and, more importantly, a significant drop in the transistor switching energy from more than 10^{-9} J to nearly 10^{-16} J [10]. Power dissipation, in a leading edge bipolar chip, with 1 cm^2 area has reached 20 - 25 W, and based on a short term extrapolation of current trends in the packaging technology, it may well be anticipated that the power dissipation might approach more than 100 watts for 50 million transistors on the same 1 cm^2 area with a switching speed of 10 ps [10]. After comparing various existing VLSI modules in terms of thermal parameters [10], the value of heat flux, Q = 25 W/cm^2 seems to be reasonable for air cooling. Considering again the critical LINK element in the MCAP, we estimate Q = 14.24 W/cm^2 to drive 2 cm of interconnect and 19 gates. It is reasonable to expect, therefore, that for a MCAP architecture implemented in MCM, air cooling would be sufficient.

5 Conclusions

The architecture to implement a class of high-performance attached processors, which can be modularly configured to match given sets of algorithms, has been presented. The high utilization rate of the processing components is achieved mainly by (1) minimizing the movement of intermediate results; (2) prefetching almost all operands using intelligent memory controllers; and (3) reconfiguring (through programming) the interconnection of the processing components to match the needs of a given algorithm.

An example MCAP architecture was evaluated for MCM implementation. Because of its commercial maturity, the CMOS technology was picked as the first (benchmark) technology to be evaluated. Transistor count for implementing the MCAP was estimated at 9.85 million. In the proposed architecture, the bottleneck is the communication through the LINK elements because of their high fan-out and relatively large interconnection distances. For the LINK element output buffers, delay, power and area calculations were made as functions of fan-out and interconnection length. For example, a LINK element with a fan-out of 19 and an average interconnection length of 2 cm has a load capacitance of 34 pF, has a delay time of 3.2 ns, occupies an area of 40,000 μm^2 and dissipates 175 mW of power. Heat flux was estimated at 14.2 W/cm^2, which leads us to believe that air cooling will be sufficient for this MCAP architecture implemented in MCM.

Further improvements in MCAP performance could be obtained by: (1) Reducing the minimal feature size to 0.5 μm or 0.2 μm, (2) Minimizing the chip to chip spacing by mounting the chips on two sides, (3) Employing a higher speed technology like GaAs (HEMT's), (4). Perhaps, using Wafer Scale Integration.

ACKNOWLEDGEMENTS

We would like to acknowledge the contributions of Stephen Senyszyn and Dan Doran to design of the MCAP, particularly those related to the instruction synchronization mechanism.

ELEMENT	DESCRIPTION	# OF TRANSISTORS
MEMORY ELEMENT	HAS 4K EACH OF RAM AND ROM	1.6M
INSTRUCTION	HAS 8 WORDS OF FIFO	16.4K
ALU	HAS 8 WORDS OF FIFO	16.4K
ELEMENTARY	HAS 8 WORDS OF FIFO	16.4K
TWO INPUT	HAS 8 WORDS OF FIFO	16.4K
JOIN	2 INPUTS AND FIFO OF 8 WORDS	16.4K
FORK	2 OUTPUTS AND FIFO OF 8 WORDS	67.4K
LINK	8 INPUTS AND 16 OUTPUTS	33.6K
STATIC RAM	16 ELEMENTS OF 1K EACH	6.3MM
SINGLE ACCESS CONTROLLER	CONTROLS 8 MEMORY ELEMENTS	11.4K
DUAL ACCESS CONTROLLER	CONTROLS 8 MEMORY ELEMENTS AND 2 DMA CHANNELS	41.4K
COMPARE	SENDS OUT FLAGS AND INDICES	33.4K
RECIPROCATE	USING CONVERGENCE METHOD	39.8K
NEGATE	INVERT THE SIGN BIT	91.6K
F.P ADDER	USING CLA'S, BARREL SHIFTERS ----	33.6K
F.P MULTIPLIER	USING MODIFIED BOOTH'S ALGORITHM	61.4K
MCAP	Total number of Transistors	6.65 Million
MCM	With 50 Chips and 300 I/O's	6.65 Million

Table 1. Transistor count for the various MCAP components

References

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1985.

[2] J. A. Swanson, G. R. Cameron, and J. C. Haberland, "Adapting the Ansys Finite-Element Analysis Program to an Attached Processor," *IEEE Computer*, vol. 16, no. 6, pp. 85–91, 1983.

[3] R. Hockney and C. Jesshope, *Parallel Computers 2*, Adam Hilger: Bristol, England, 1988.

[4] T Hoshino, *PAX Computer: High-Speed Parallel Processing and Scientific Computing*, Addison-Wesley Publishing Company: Reading, Massachusetts, 1985.

[5] J. H. Tang and E. S. Davidson, "An Evaluation of Cray I and Cray X-MP Performance on Vectorizable Livermore FORTRAN Kernels," *Proc. 1984 Int'l Conf. on Supercomputing*, pp. 510–518, 1988.

[6] J. Dongarra, "Linear Algebra Libraries for High-performance Computers: A Personal Perspective," *IEEE Parallel & Distributed Technology*, pp. 17–24, February 1993.

[7] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, "NAS Parallel Benchmark Results," *IEEE Parallel & Distributed Technology*, pp. 43–51, February 1993.

[8] H. Nakano and et. al., "An 80 MFLOPS (peak) 64-bit microprocessor for parallel computer," *IEEE J. Solid-State Circuits*, vol. SC-27, no. 3, pp. 365–372, 1992.

[9] N. Weste and K. Eshraghian, *Principles of CMOS VLSI design - A systems perspective*, Addison Wesley, 1993.

[10] A. B. Cohen, "Thermal management of air and liquid cooled MCM's," *IEEE Trans on Components, Hybrids, Manufacturing Technology*, vol. CHMT-10, no. 2, pp. 159–175, 1987.

ATTACHMENT B

SIMULATION AND PERFORMANCE EVALUATION OF A MODULARLY CONFIGURABLE ATTACHED PROCESSOR

Yi-Chieh Chang, Glenn Gibson, and Claudia Ayala

Department of Electrical and Computer Engineering
The University of Texas at El Paso
El Paso, Texas 79968-0523

Abstract

A new architecture for high-performance parallel attached processors is studied in this paper. The unique features are that the attached processor can be configured to match a set of algorithms and its memory controllers can be programmed to fit the access patterns required by the algorithms. As a result, high utilization of the processing logic for given sets of algorithms can be obtained. A simulator with interactive graphic interface is designed to study the performance of the proposed architecture. An example based on matrix multiplication is used for illustration. The simulation results show that a sustained execution rate as high as 95% of the peak speed for matrices with a size of 128×128 can be achieved in the proposed attached processor architecture. If CMOS technology is chosen to implement the MCAP architecture, a sustained speed of 190 MFLOPS can be obtained for matrix multiplication with four multipliers and four adders. [1]

1 Introduction

An attached, or back-end, processor is a processing system that is connected to a host computer for the purpose of very quickly executing most of the overall system's computational tasks. In such an organization, "the host is a program manager which handles all I/O, code compiling, and operating system functions, while the back-end attached processor concentrates on arithmetic computation with data supplied by the host machine" [1].

The specific purpose of an attached processor is to execute members of a set of algorithms very quickly. The broader the set of algorithms the more generally applicable the attached processor. The underlying goal of the designer is to efficiently utilize the hardware for as broad a set of algorithms as possible. However, for most current designs, the average sustainable execution rates have been found to be only 5% to 20% of their peak rates, which are determined by summing the maximum computational rates of the processing elements. For example, the sustainable rate for a Cray X-MP with four processors may be as low as 5% for some algorithms [2]. Also extensive evaluations of recent high-performance computations using Lapack are given in [3] and using NSA parallel benchmarks are given in [4]. These evaluations confirm the low efficiencies of most supercomputers. Although some of the lost efficiency is necessitated by the algorithms, much of it is due to memory accessing and contention for shared resou. . in general, including internal buses.

Described in this paper is a class of high-performance attached processors called Modularly Configurabie Attached Processors (MCAPs) which can attain quickness and high utilization through:

- Closely matching their architectures to the set of algorithms they are to execute.
- Overlapping of processing and memory accessing by using memory prefetching.
- Minimizing the movement of data.
- Using a high-speed technology with MCM or wafer scale implementations.

An MCAP is constructed from the component types specified in Sec. 2. These component types are such that each member of the class may include parallel processing and memory-to-memory pipelines, and be constructed in a building block fashion. They encompass routing components (including buses) as well as memory, control, and processing components. By overlapping processing with memory accessing and matching an architecture with a set of algorithms, it is predicted that the average sustainable rate for a specific set of algorithms can attain at least 60% of the peak rate. By defining components that are simple enough to be fabricated onto single low-density ICs, a high-speed technology may be used.

In order to study and analyze the performance of the MCAP architecture, a set of simulation tools has been developed. These tools include an architecture editor, an assembler, and a simulator. The main objective of this paper is to study the performance of the MCAP architecture

[1] The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency.

using the developed simulation tools. Matrix multiplication is studied as an example to illustrate how to match an algorithm to a specific MCAP architecture. Through the simulation, the hardware attributes of each component, such as the execution delay and the size of the data queue can be fine tuned to achieve a high sustained execution rate. For example, the simulation results showed that for matrices with a size of 128× 128, a sustained rate as high as 95% of the peak speed can be achieved.

The rest of this paper is organized as follows. Section 2 briefly describes the architecture of the MCAP and the fundamental components required to construct an MCAP. Section 3 describes the simulation tools developed for the performance analysis of the MCAP architectures. Section 4 shows how to design a simulation program to match an algorithm on a given MCAP architecture. Section 5 shows the simulation results and discusses the various design principles for achieving a high sustained execution speeds on MCAP architectures. This paper concludes with Section 6.

2 MCAP Architecture

An MCAP is an attached processor that is constructed entirely from a standard set of connections and components. This standard set consists of three types of asynchronous connections and twelve types of components. The definitions of the connection and component types provide a standard set of rules that allow the components to be easily configured in different ways to construct attached processors that can efficiently perform different sets of algorithms.

An MCAP operates by drawing an instruction stream from the memory component into the instruction component. The instruction component uses internal instructions in the stream to form external instructions that are then distributed to the other non-memory components through the MCAP's bus component. All components in the instruction stream include input instruction queues. When the non-memory components have received all of the instructions needed to perform an algorithm, they automatically prefetch the data from the memory components, route the data to and from the processor components and store the results back into the memory components. All non-memory components have input data queues. DMA units built into some controller components, which are the components that supervise all memory accessing, are used to automatically transfer data between the host's main memory and the MCAP's memory components while the algorithm is executing. Also, the instruction and data streams are separate, thereby allowing the instructions needed for the next algorithm to be distributed while the current algorithm is executing.

An example architecture is given in Fig. 1. Its processing subsection includes a comparator, a negator (elementary component), a reciprocator (elementary), a set of pipelined adders capable of accumulation, and a set of pipelined multipliers. Each adder or multiplier is constructed of four stages pipeline [a two-input (T) component followed by three elementary (E) components]. All communications to and from the processing components are through six link components, three on each side of the processor. Join and fork components are provided to allow flexible use of the link components. Also, to allow for accumulation there is a feedback connection between the fork component at the output from each adder and the join component at the input to the adder. There is a dual-access component to provide intermediate memory and a connection to main memory. The single-access component provides internal storage. The detail description of these basic components can be found in [5].

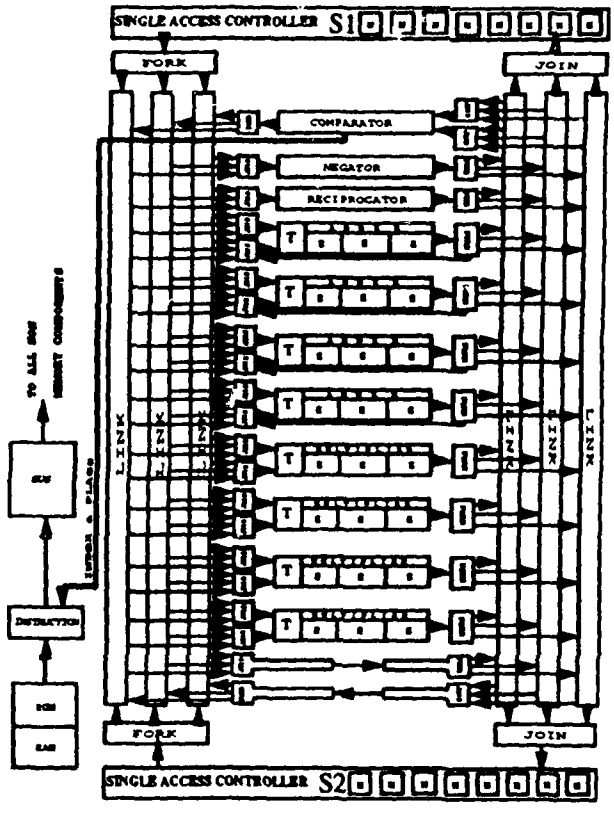

Fig. 1. An example MCAP architecture.

3 Simulation tools for the performance analysis of the MCAP architectures

Three CAD tools have been developed for the performance analysis of the MCAP architectures. These tools include an architecture editor, an assembler, and a simulator. All these tools are written in C++ and installed on a PC compatible with a 486 microprocessor.

3.1 Architecture editor

An interactive graphics editor is designed to facilitate the construction of an MCAP architecture. The architecture editor provides the following functions for constructing an MCAP architecture: (1) Creating, deleting, and moving

around any fundamental component defined in Section 2 in an existing architecture, (2) Adding or deleting a connection between any two components, (3) Modifying the attributes of any component, such as the execution time, instruction or data queue size, number of pipeline stages, capacity, etc. All the above functions are performed on an interactive graphic display, thus it is very easy and convenient to construct any kind of MCAP architecture. The output file generated by the architecture editor is the architecture source file which is later used by the assembler and the simulator. The architecture source file specifies the detailed information of an MCAP architecture, such as the component ID, the number of connections and their connection numbers, and other attributes of each component.

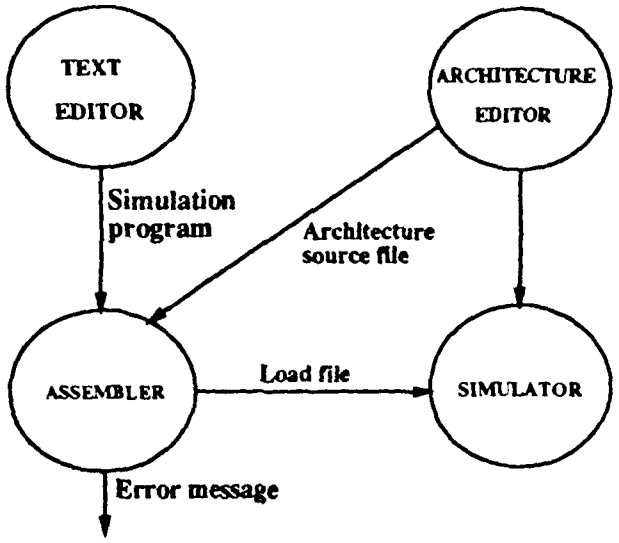

Fig. 2. The simulation tools developed for simulating the MCAP architecture.

3.2 Generation of a load file for the graphic simulator

The assembler compiles a simulation program and an architecture source file, then generates a load file to be used by the simulator. The assembler runs two passes. In the first pass, the assembler reads the architecture source file and builds two tables. The first table associates the component's mnemonic to its corresponding ID number, and the second table lists all the connections along with their corresponding source and destination components. During the first pass, the assembler also includes all defined symbols and their values in the first table. In the second pass, the assembler reads the simulation program again and produces the load statements for each instruction. If the simulation program references only components listed in the architecture file, no syntax errors are generated and the assembler produces a load file; otherwise, the assembler produces an error file (.SLT) which lists the line numbers in the simulation program where the errors occur. A segment of the load file for the link component, L80, is shown below.

```
53  L80  52    ;set the mode
57  L80  1 1 135    ;set input connection
59  L80  4 4 151 196 190    ;set the output connections
58  L80  32 5 32767 151 196 190    ;set output patterns
51  L80  33    ;set the number of output operands
```

Fig. 2 shows the process of creating a load file from the *simulation program* and an architecture source file.

3.3 MCAP Simulator

The simulator is designed to simulate the operation of each component in an MCAP architecture while an algorithm is being executed on the architecture. This allows an architecture to be matched to an algorithm. The structure of the state diagrams for the memory, instruction and bus components are shown in Fig. 3. Note that each component may take on a subset of the following states:

FREE — there is no activity in the component
DIST — the component is waiting for an instruction to distribute
DBSY — an instruction is being distributed to its register(s)
IDLE — the component is waiting for input
BUSY — the component is being executed
WAIT — the component is waiting for its output to be taken

The simulator first brings in the architecture source file and the program to be simulated, opens a result file, asks the user for the format of the results and then begins the simulation. The pseudocode for the simulator is

```
Retrieve architecture source file
Retrieve simulation program file
Open results file and request the format of the results
Initialize variables (includes setting the system time
to zero)
DO {
    Update components in BUSY state
    Update components in WAIT state
    Update instruction and data queues
    Update components in IDLE state
    Update components in FREE state
    Update components in DIST state
    Update components in DBSY state
    Increment system time
    If format requires results to be stored, then output
    results
} While (not end of simulation)
Close results file
```

Inside the Do loop, the simulator first updates all components currently in the BUSY state. Their execution times,

which are set to their maximum values when the BUSY state is entered, are decremented and, for those that become zero, the component's state is changed (usually to the IDLE state, see Fig. 3). As the transition occurs, appropriate actions are taken. Similarly, the components in the other states are checked. If the transition conditions are met (e.g., the output has been accepted by the succeeding component), then appropriate actions are taken and the component is put into its next state. The simulator is updated such that a component can have at most one state change each time around the loop. Also, all instructions and data queues are updated each time around the loop. The results primarily consist of the times each component spends in each of its states and are recorded at the times specified by the results format.

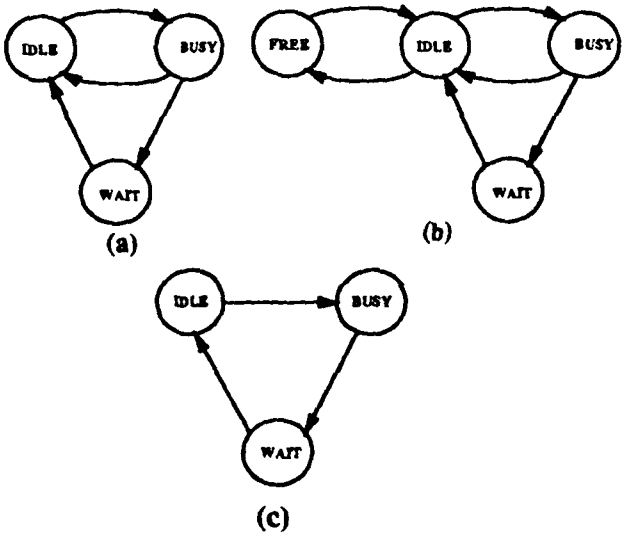

Fig. 3. State diagram structure for the (a) memory, (b) instruction, and (c) bus components.

4 Matching an algorithm to an MCAP architecture

In order to efficiently use the available logic and interconnections, an architecture must be carefully matched to an algorithm or set of algorithms. This involves a study relating the flows, storage and processing of the data required by the algorithm(s). Clearly, there is no point in increasing the speed of a processing subsystem if the current interconnections and memory hierarchy are inadequate to support the processing (or vice versa). But a good balance for one algorithm may not be a good balance for a different algorithm. What is needed is a satisfactory tradeoff for the work mix expected of a system and a means of evaluating the design parameters chosen.

Space allows only a single example, so let us consider the computation that most frequently occurs in computationally intense algorithms, matrix multiplication. Let us examine how the MCAP in Fig. 1 could be analyzed relative to the algorithm $AB = C$ using the middle product method [6] where A, B and C are $n \times n$ matrices. Assume the S2's memory can hold the entire matrices for the multiplication.

The algorithm consists of the computations

$$\sum_{i=1}^{n} a_{ij} B_j = C_i \qquad i = 1, \ldots, n$$

where the a_{ij}s are the elements of A, the B_js are the rows of B, and the C_is are the rows of C. The algorithm proceeds by storing the first n elements of the first column of A and the first n rows of B in the S2's memory. Then the products $a_{i1}B_1$, for $i = 1, \ldots, n$, are formed and stored in the S1's memory. Next, the first n elements of the second column of A are brought into the S1 and the products $a_{i2}B_i$ are formed and added to the corresponding previous products, with the results being returned to the S1.

By matching this algorithm with the architecture in Fig. 1, it is seen that each adder and multiplier must perform approximately $n^3/2$ operations and each link on the left and two of the links on the right must perform approximately n^3 transfers. (The third link on the right is not be needed.) The approximate number of accesses to the S components is about $2n^3$. If T is the per stage processing time of the multipliers, then T should also be the per stage processing time of the adders and $T/4$ should be the transfer time of the links. The access times of the S components should be $T/8$ for both reads and writes. For $T = 40$ ns, the link transfer time should be 10ns and the average memory access times should be 5 ns. The computation rate would be 200 Mflops per second. If the MCAP were put into an MCM or wafer and memory interleaving were used, these times would be within the capability of current HCMOS technology.

4.1 Design of a simulation program for an MCAP Architecture

To verify the above simple analysis, we have designed a set of simulation programs for the matrix multiplication to be run by the MCAP simulator. The instruction set for an MCAP architecture consists of two sets of instructions, internal and external. The former is processed within the instruction component and the latter is distributed by the instruction component to the corresponding components. In this paper we will only discuss the external instruction set. The external instructions set consists of three types of instructions: instructions which set the number of operands to be output from or input to a component, instructions which set the mode of a component, and instructions which set input or output connection patterns for the router components or partition and operand patterns for the controller components.

When programming an MCAP architecture, each component must be programmed individually. The operation mode, input/output patterns, connections, number of operands to be input or output, broadcasting patterns, etc., are programmed for each component using external

instructions. For example, referring to Fig. 1, the input data stream is supplied from the S2 component. This data stream passes through link component and is distributed among four multipliers (each multiplier is composed of one T and three E components), then the output of the multipliers is sent back to the four adders for accumulating the results. The S1 component stores the intermediate results output from the adders and sends back to the adders when new products are produced from the multipliers. The final results are stored back to S2 component. For the matrix multiplication discussed in the previous section as an example, the instruction set for programming each component is discussed below.

4.2 Programming the single access, S, component

To compute one product for one row of the C matrix, one needs to broadcast one element of A and then transfer one row of B to the multipliers. Since there are n products to be multiplied and added to form a row of C, this operation needs to be repeated n times. So, the total number of operands to be output from the S2 components to the multipliers is $n^2(n+1)$. The memory access controller, i.e. S component, connects to one or more memory modules and allows the storage and retrieval of data to and from these modules. Memory as a whole is divided into a specified number of partitions across the modules. Thus, the consecutive addresses of a single partition extend over adjacent modules to reduce the average memory access time. Also, partitions allows the data to be stored in a particular pattern according to the requirement of the algorithm. Fig. 4 shows how the partitions are distributed across the memory modules.

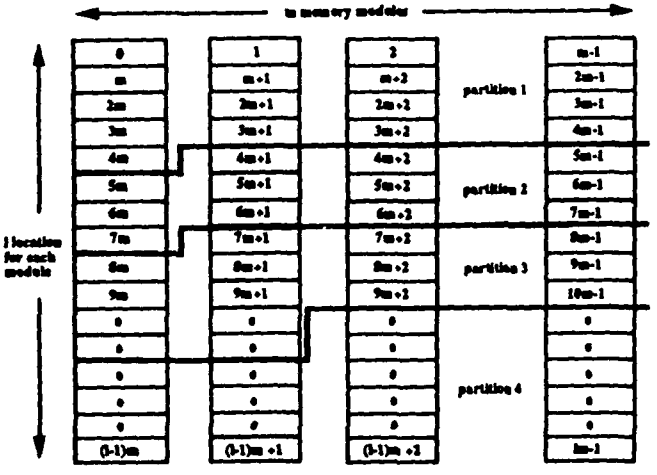

Fig. 4. Distribution of partitions across memory modules.

To access a memory location, the S component alternates between the partitions listed in the programmed partition pattern. As an example, to store two $n \times n$ matrices A and B, we divide the memory connected to S2 in the example architecture into two partitions using the 'spbs' (S partition base and size) instruction: Thus, the instruction sequence for the S2 component is shown below:

```
smod S2, mode   ;set mode
sopp S2, #n, 0, 1  ;set output pattern, 1 followed by n
spbs S2, 0, 0, N  ;set base address for partition 0
spbs S2, 1, N, N  ;set base address for partition 1
stko S2, N×(n+1) ;set the number of output operands
```

The first instruction configures the S2 component for output only and sets the output pattern of partitions 0 and 1 to a 1 then n pattern. Since the S component only outputs data, only the output partition pattern needs to be specified. The second instruction generates a pattern in which partition 0 is accessed once and partition 1 is accessed n times. This pattern is repeated N times. The third instruction defines partition 0 to start at base address 0 and assigns it a size of N. The fourth instruction defines partition 1 to start at base address N and assigns it a size of N. Using the middle product method for the matrix multiplication, the first element of the first row of matrix A needs to be output first followed by the first row of matrix B. Then, the second element of the first row of matrix A is output followed by the second row of matrix B and so on.

Using the instructions shown above, S2 will output the data stream $a_{11}, b_{11}, \ldots, b_{1n}, a_{12}, b_{21}, \ldots, b_{2n}, \ldots, a_{1n}, b_{n1}, \ldots, b_{nn}$ to produce the first row of C. Similarly, S2 will output the data stream stream $a_{21}, b_{11}, \ldots, b_{1n}, a_{22}, b_{21}, \ldots, b_{2n}, \ldots, a_{2n}, b_{n1}, \ldots, b_{nn}$ to produce the second row of C, etc.

4.3 Programming the link component

The operation of the link component, LINK, consists of the broadcasting of one element of matrix A and distributing a row of matrix B among four join components, for example J059, J057, J060, and J062. Thus, the number of operands to be output by LINK is N (1 + n). The following instructions program the LINK component for the matrix multiplication:

```
lmod LINK, mode   ;set mode
lsip LINK, F077   ;set input pattern
lsbp LINK, J059, J057, J060, j062 ;set broadcast pattern
lsop LINK, #n, &, J059, J057, J060, j062 ;output pattern
ltko LINK, NumOpsOutL   ;set number of output operands
```

The third instruction sets the broadcasting pattern which specifies the components to receive the broadcast operand. The fourth instruction programs the LINK component to perform a broadcast, indicated by the symbol &, and then to distribute n operands to the components listed in the instruction. The # character always precedes the value of n when using a 1 then n pattern or an n then 1 pattern.

4.4 Programming the, Join, Fork, and Processing components

The Fork, Join, and processing components, T and E, can be programmed in a similar fashion. Since each multiplier

and adder is composed of four pipeline stages, one T component followed by three E components are needed to configure one multiplier or one adder. The first pipeline stage for the processing component must use the T component because the T component can take two input operands while the E component can only take one input operand from the previous pipeline stage.

5 Simulation and performance study

The sustained speed for executing the matrix multiplication on the MCAP architecture shown in Fig. 1 is studied through the simulation. The execution time for each component is determined from an earlier paper [5] where an MCM implementation of the MCAP architecture using CMOS technology was studied and analyzed. From this paper, we found that the total number of transistors needed for implementing the MCAP architecture shown in Fig. 1 is at around 9.85 million transistors and the whole system can be built into a 5 cm by 5 cm MCM package using CMOS technology. Based on this study, the estimated execution time for each component is listed below [5]:

- Multiplier and adder (64-bit floating-point processor): 40 ns per pipeline stage.
- Link components: 8 to 10 ns
- Join and fork components: 4 to 6 ns
- S component (memory controller): 4 to 8 ns for execution time, address generation time, and bus access time
- Memory module: 80 ns (two-port DRAM)

Several simulations were run to study the best obtainable sustained speed for matrix multiplication on the MCAP architecture. The sustained rate compared to the peak speed vs. matrix size is plotted in Fig. 5 and the actual MFLOPS vs. matrix size is plotted in Fig. 6. The total clock cycles and the average percentage of BUSY, WAIT, IDLE, and FREE states for the multipliers and adders is given in Table 1. Many interesting results can be derived from the simulation data. Note that the peak speed of the MCAP architecture shown in Fig. 1 is at 200 MFLOPS with four multipliers and four adders using the above execution time derived from [5].

5.1 Sustained rate in the MCAP architecture

In order to achieve a very high sustained rate in an attached processor, the bottleneck component needs to be identified, so that the system performance can be improved. From the simulation study, the bottleneck component in the MCAP architecture moves from one component to another component when the address generation time for the S component is reduced.

For example, the lower two curves in Fig. 5 showed the sustained rate remains unchanged when the address generation time for S004 is reduced from 6 ns to 4 ns while the execution time for all other components is unchanged. If the address generation time of S004 is set at 6 ns while reducing the execution time of other components, such as the Join, Fork, or Link components, no improvement on the sustained rate was found. However, if the address generation time of S004 is set at 4 ns and the execution time of J006 is reduced from 6 ns to 5 ns, the highest sustained rate is increased from 83.3% to 90.9% as shown in Fig. 5. Further reducing the execution time of [illegible] to 4 ns increases the sustained rate to 94.6% fo[illegible]ces with a size of 128 × 128. Also, if the address g[illegible]ion time of S004 is fixed between 4 ns and 6 ns, no improvement on the sustained rate was found by reducing the memory access time.

Based on the simulation data collected for matrices with sizes ranging from 4 × 4 to 128 × 128, we are able to predict the sustained rate for larger matrices. The simulation results and estimated data are given in Table 1. From this table, the sustained rate is estimated to be as high as 96% for matrices with a size of 1024 × 1024 or larger.

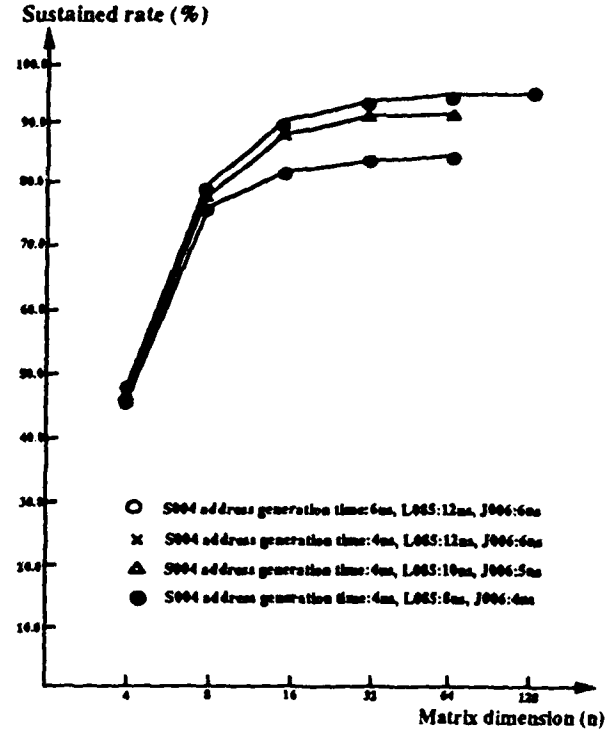

Fig. 5. Sustained rate vs. matrix size for matrix multiplication.

5.2 Performance comparison with other high-performance parallel computers

In one of the recent issues of *IEEE Parallel and Distributed Technology*, a thorough performance comparison between various high-performance parallel computers was made using the LinPack benchmark [3]. Since matrix multiplication is the main portion of the computation in LinPack

benchmark, it is appropriate to compare the simulation results obtained in this study to the results reported in [3]. In [3], it showed that Cray X-MP/1 with peak speed at 235 MFLOPS achieves the highest sustained rate at 51% (121 MFLOPS actual speed) while Cray C-90/16 can deliver the highest actual speed at 479 MFLOPS but with only 3.1% of its peak speed of 15,238 MFLOPS.

In the MCAP architecture, the peak speed for four multipliers and four adders is at 200 MFLOPS using CMOS technology. However, by matching the algorithm to the architecture, the best sustained rate can be as high as 94.6% for matrices with a size of 128×128. The actual speed obtainable from the MCAP architecture is at 190 MFLOPS which is within the same range as Cray Y-MP/1 (145 MFLOPS) or X-MP/4 (178 MFLOPS). Note that both Cray computers are designed using the ECL logic and the cost for both machines is several million dollars, while the MCAP architecture is based on the much cheaper CMOS technology.

Comparing with microcomputer or workstations, MCAP has the following advantages. First, it is very easy to construct an MCAP architecture to match an algorithm so that a high sustained rate can be obtained. Second, the S component can be programmed in ahead of time for a new algorithm before completing the current algorithm, so that the instruction fetching time can be overlapped with the execution of the current algorithm. Third, the MCAP architecture can be scaled up to include 10 to 20 processing components to achieve a peak performance between 200 to 500 MFLOPS using CMOS technology. Lastly, the MCAP can be constructed from a few basic components and it's architecture is much simpler than any modern microprocessor or workstations.

6 Conclusions

The architecture to implement a class of high-performance attached processors, which can be modularly configured to match given sets of algorithms, has been presented. The high utilization rate of the processing components is achieved mainly by (1) minimizing the movement of intermediate results; (2) prefetching almost all operands using intelligent memory controllers; and (3) reconfiguring (through programming) the interconnection of the processing components to match the needs of a given algorithm. Moreover, using the small set of fundamental components defined for the MCAP architecture, it is possible to quickly prototype an MCAP architecture tailored for a group of specific applications.

A set of simulation tools has been developed to evaluate the performance of the MCAP architecture. Through simulation studies, we found that it is possible to achieve a very high sustained rate by matching the algorithm to the MCAP architecture. Thus, although the peak speed of the MCAP architecture may not be very high, the actual sustained speed obtainable on the MCAP architecture is in the same range of many supercomputers such as, the Cray X-MP/1 or Y-MP/4. Furthermore, the power consumption on the MCAP architecture is estimated at 25 W/cm^2 which is suitable for air cooling [5]; this is in sharp contrast to most supercomputers where liquid cooling is required.

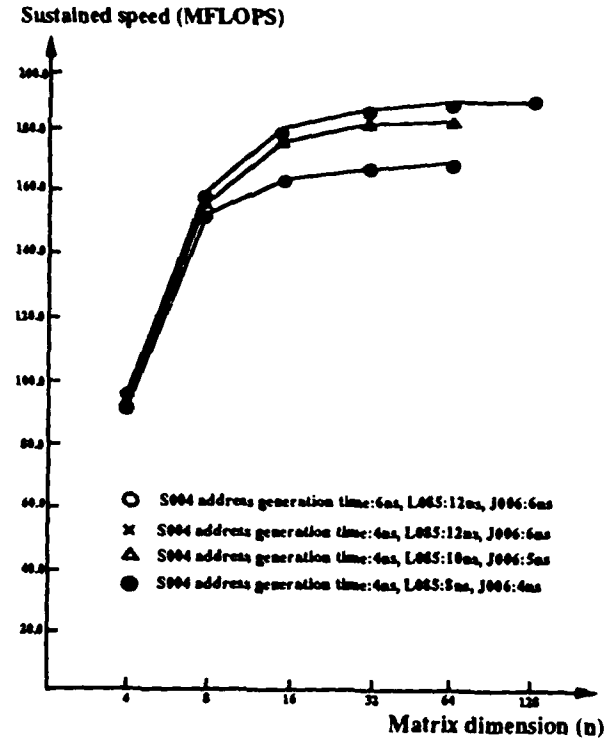

Fig. 6. Sustained speed vs. matrix size for matrix multiplication.

References

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1985.

[2] J. H. Tang and E. S. Davidson, "An Evaluation of Cray I and Cray X-MP Performance on Vectorizable Livermore FORTRAN Kernels," *Proc. 1984 Int'l Conf. on Supercomputing*, pp. 510–518, 1988.

[3] J. Dongarra, "Linear Algebra Libraries for High-performance Computers: A Personal Perspective," *IEEE Parallel & Distributed Technology*, pp. 17–24, February 1993.

[4] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, "NAS Parallel Benchmark Results," *IEEE Parallel & Distributed Technology*, pp. 43–51, February 1993.

[5] G. Gibson, , V. Singh, S. Singh, Y. C. Liu, Y. C. Chang, and S. Cabrera, "MCM Implementation of Modularly Configurable Attached Processors," *Submitted for publication in 7th Int'l Conf. on Parallel and Distributed Computing Systems*, 1994.

[6] R. Hockney and C. Jesshope, *Parallel Computers 2*, Adam Hilger: Bristol, England, 1988.

ATTACHMENT C

APPLICATION OF COMPUTATIONAL INTENSITY TO MEMORY HIERARCHY DESIGN

Glenn Gibson, Yu-cheng Liu, Yi-Chieh Chang

Department of Electrical and Computer Engineering
The University of Texas at El Paso
El Paso, Texas 79968-0523

Abstract

The computational intensity of the task being executed is an important factor in determining the sustainable throughput, especially for modern computers with hierarchical memories and highly pipelined processors. This paper determines the computational intensity with respect to the inner memory capacity for several computationally intensive algorithms that have wide application. It also analyzes the influences of computational intensity on the speed and cost of hierarchical memories. Based on the analysis, a method to optimize the memory cost relative to the memory size and speed at each memory level is also presented. [1]

1 Introduction

The performance of a computer system is typically limited by the speed of its memory and cache memory has been widely used in reducing the memory access time [1], [2], [3]. In the two-level memory hierarchy shown in Fig. 1, where memory accesses must be made through the first-level memory, M is the number of locations in outer memory, m is the number of locations in inner memory, and n_p is the number of floating-point operations involved in the algorithm.

The average access time is $t_{ave} = t_m + (1-h)t_M$, where h is the hit ratio, and t_m and t_M are the access times of the inner and outer memory, respectively. The inner memory of this two-level hierarchy can be implemented in a variety of ways and may represent a memory subhierarchy. Obviously the hit ratio increases as the size of the first-level memory grows. However, in addition to size, the hit ratio is also a function of the algorithms and applications involved. An algorithm that exhibits a high degree of *locality in memory* accesses will result in a high hit ratio. For computation-intensive algorithms, this programming locality can be measured as the computational intensity, which is defined as the average number of floating-point operations per access to the second level memory.

[1] The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency.

Another reason to investigate computational intensity is that it directly affects the average performance of a pipelined processor. For a highly pipelined machine, main memory becomes the bottleneck of the system. The speed imbalance between the memory and processing elements causes the average performance to be substantially lower than the peak performance designed into the processor. Sustainable computation rates of typical supercomputers have been evaluated and reported by Tang and Davidson [4] and found to be much lower than the sustainable rates for most algorithms. A high computational intensity reduces the rate of memory access, thus improving the average performance. Hockney and Jesshope [5] have shown the average performance as a function of the computational intensity for nonoverlapped as well as overlapped memory transfer and arithmetic operation.

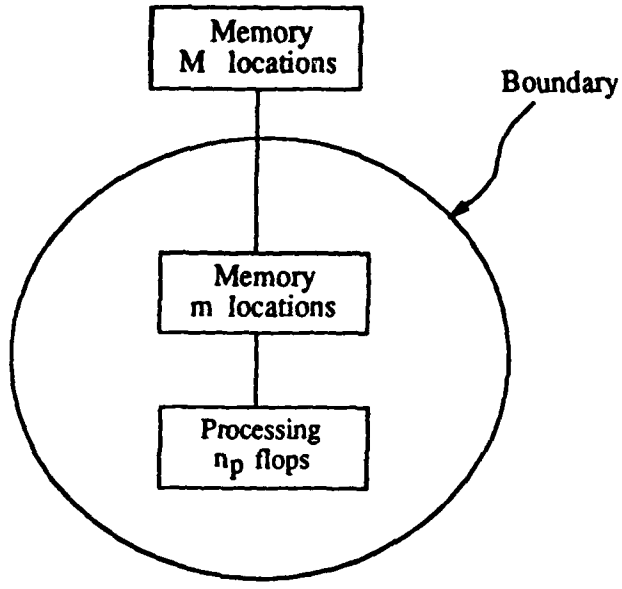

Figure 1. Two-level memory hierarchy structure.

In this paper, we investigate the computational intensities for some of the frequently used algorithms and their impact on the design of hierarchical memory

systems. Section 2 analyzes several computationally intensive algorithms, including matrix multiplication, matrix inversion, and solutions to linear and partial differential equations. For these algorithms, analytical methods are developed to evaluate their computational intensities with the inner memory capacity and problem size as the parameters. Sections 3 and 4 consider possible applications of computational intensities in the design of hierarchical memory systems. Expressions for optimizing a hierarchy with respect to speed and cost are given in terms of computational intensities.

2 Case Studies of Computational Intensities

In this section the computational intensities as functions of internal memory size and problem size are determined for several example algorithms. When $m = 0$, only the pipeline buffer registers are available for internal storage. In some cases, chaining of processing components is assumed. In deriving the required number of accesses to the outer memory level it is assumed that the inner memory is entirely usable (e.g., it is fully associative).

2.1 Matrix Multiplication

As discussed in [5], there are three major algorithms for matrix multiplication, the inner product, middle product, and outer product algorithms. All three algorithms require the same number of operations. However, the sequences of computation are different for the three approaches.

The analysis assumes that each matrix is an $n \times n$ matrix and the multiplication is:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ & & \cdots & \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ & & \cdots & \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

Hence, c_{ij} is computed as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

and the total number of computations is $n \times n \times (2n-1) = 2n^3 - n^2$, which is the same for all three algorithms.

The inner product method computes the elements of C in sequence. This is implemented in a high-level languages as

```
for i = 1 to n do
    for j=1 to n do
        for k =1 to n do
            C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

The sequence of computations is $c_{11} = a_{11} \times b_{11}$, $c_{11} = c_{11} + a_{12} \times b_{21}$, $c_{11} = c_{11} + a_{13} \times b_{31}$, ..., $c_{11} = c_{11} + a_{1n} \times b_{n1}$, $c_{12} = a_{11} \times b_{12}$, $c_{12} = c_{12} + a_{12} \times b_{22}$, ..., $c_{12} = c_{12} + a_{1n} \times b_{n2}$, ...

Since this method computes one element c_{ij}, i.e., one inner product, before computing the next element, it is advantageous to keep c_{ij} in the inner memory to reduce accesses to the outer memory. Also shown in the above computation sequence is that in computing each row of matrix C, all elements in matrix B are referenced once. On the other hand, in the same computation, only one row of matrix A is referenced and it is used n times. Therefore, when the available storage m is between 1 and $n+1$, the best possible allocation is to store the matrix C elements currently being computed and $m-1$ matrix A elements. Based on this, the computation of each row in matrix C requires n^2 fetches and n stores. For a row of c's, there are $m-1$ fetches for the a's kept in the inner memory plus $(n-m+1)n$ fetches for the remaining $n-m+1$ a's from the outer memory. The total number of memory accesses is $[(n^2+n+m-1+(n-m+1)n]n = 2n^3 + 2n^2 - mn^2 + mn - n$. Thus, the computational intensity is $(2n^3 - n^2)/(2n^3 + 2n^2 - mn^2 + mn - n)$.

Once the storage exceeds $n+1$, there is no advantage in storing more than one row of a's. This is because after one row of c's is computed, the same row of a's will not be used again. Therefore, when the inner storage size is between $n+1$ and $(n+1)n+1$, the inner memory should keep one c, one row of a's and as many columns of b's as the remaining space allows. These columns of b's will be used in computing each row of c's. For the case $kn+n+1$, k columns of b's can be kept in the inner memory. The total number of accesses is $n^2 + n^2 + kn + (n-k)n^2 = n^3 + 2n^2 - kn^2 + kn$. This includes n^2 fetches of a's, n^2 stores of c's, kn fetches of b's, that are kept in the inner memory plus $(n-k)n^2$ fetches for the remaining b's. Table 1 summarizes computational intensities for various storage sizes.

m	Computational intensity
0	$\frac{2n^3-n^2}{4n^3-n^2} \approx \frac{1}{2}$
1	$\frac{2n^3-n^2}{2n^3+n^2} \approx 1$
$1 < m < n+1$	$\frac{2n^3-n^2}{2n^3+2n^2-mn^2+mn-n} \approx 1$
$n+1$	$\frac{2n^3-n^2}{n^3+n^2} \approx 2$
$2n+1$	$\frac{2n^3-n^2}{n^3+n^2+n} \approx 2$
$kn+n+1$	
$k = 3, 4, \ldots, n-1$	$\frac{2n^3-n^2}{n^3+2n^2-kn^2+kn} \approx 2$
$(n+1)n+1$	$\frac{2n^3-n^2}{3n^2} \approx \frac{2n}{3}$
$3n^2$	$\frac{2n^3-n^2}{3n^2} \approx \frac{2n}{3}$

Table 1 Computational intensities for the inner product method.

The middle product method computes an entire column of c's simultaneously, thus allowing up to n pro-

cessors to compute in parallel. In a high-level language, this can be implemented as

```
for j = 1 to n do
    for k=1 to n do
        for i =1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j];
```

The sequence of computations becomes $c_{1j} = a_{11} \times b_{1j}$, $c_{2j} = a_{21} \times b_{1j}$, $c_{3j} = a_{31} \times b_{1j}$, ..., $c_{nj} = a_{n1} \times b_{1j}$, $c_{1j} = c_{1j} + a_{12} \times b_{2j}$, $c_{2j} = c_{2j} + a_{22} \times b_{2j}$, $c_{3j} = c_{3j} + a_{32} \times b_{2j}$, ...

Since each b_{kj} is used n times and may then be discarded, keeping b_{kj} in inner memory will reduce the number of memory accesses. For $1 < m < n+1$, the remaining locations store $m-1$ c's. Based on this organization, the total number of memory accesses required in computing one column of c's is $n(n+2)+2(n-m+1)(n-1)$. The first term includes n writes for storing a column of matrix C. The second term represents the number of accesses required to save and retrieve partial sums for the remaining $n-m+1$ c's that are not kept in the inner memory. Therefore, the entire matrix multiplication requires $n(3n^2+2n-2mn+2m-2)$ memory accesses.

For a memory size of $kn+n+1$, in addition to one b and one column of c's, k columns of a's can be kept in memory. This leads to the following total number of accesses for the entire matrix multiplication:

$$n^2 + n^2 + kn + (n-k)n^2$$

The first term is to fetch b's, the second term to store c's, the third term to fetch k columns of a's and the last term to fetch the remaining a's n times. Computational intensities for the middle product algorithm are summarized in Table 2.

m	Computational intensity
0	$\frac{2n^3-n^2}{4n^3-n^2} \approx \frac{1}{2}$
1	$\frac{2n^3-n^2}{3n^3} \approx \frac{2}{3}$
$1 < m < n+1$	$\frac{2n^3-n^2}{3n^3+2n^2-2mn^2+2mn-2n} \approx \frac{2}{3}$
$n+1$	$\frac{2n^3-n^2}{n^3+2n^2} \approx 2$
$2n+1$	$\frac{2n^3-n^2}{n^3+n^2+n} \approx 2$
$kn+n+1$	
$k = 3, 4, \ldots, n-1$	$\frac{2n^3-n^2}{n^3+2n^2-kn^2+kn} \approx 2$
$(n+1)n+1$	$\frac{2n^3-n^2}{3n^2} \approx \frac{2n}{3}$
$3n^2$	$\frac{2n^3-n^2}{3n^2} \approx \frac{2n}{3}$

Table 2 Computational intensities for the middle product method.

The outer product algorithm further increases the degree of parallelism by computing all n^2 c elements at the same time. This allows up to n^2 processors to compute the matrix multiplication in parallel. In a high-level language, this algorithm is implemented as

```
for k = 1 to n do
    for i=1 to n do
        for j =1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j];
```

The sequence of computations becomes $c_{11} = c_{11} + a_{1k} \times b_{k1}$, $c_{12} = c_{12} + a_{1k} \times b_{k2}$, $c_{13} = c_{13} + a_{1k} \times b_{k3}$, ..., $c_{1n} = c_{1n} + a_{1k} \times b_{kn}$, $c_{21} = c_{21} + a_{2k} \times b_{k1}$, $c_{22} = c_{22} + a_{2k} \times b_{k2}$, ...

For $1 < m < n+1$, one a and $m-1$ b's should be internally stored. This leads to the total number of accesses $3n^3-(m-1)n^2+(m-1)n$. For a memory size of $kn+n+1$, the inner memory should store one a, one row of b's and k rows of c's. The resulting number of memory accesses in this case is $2n^3+n^2-2kn^2+2kn$. Table 3 shows computational intensities for various memory sizes based on the outer product algorithm.

m	Computational intensity
0	$\frac{2n^3-n^2}{4n^3-n^2} \approx \frac{1}{2}$
1	$\frac{2n^3-n^2}{3n^3} \approx \frac{2}{3}$
$1 < m < n$	$\frac{2n^3-n^2}{3n^3+n^2-mn^2+mn-n} \approx \frac{2}{3}$
$n+1$	$\frac{2n^3-n^2}{2n^3+n^2} \approx 1$
$2n+1$	$\frac{2n^3-n^2}{2n^3-n^2+2n} \approx 1$
$kn+n+1$	
$k = 3, 4, \ldots, n-1$	$\frac{2n^3-n^2}{2n^3+n^2-2kn^2+2kn}$
$(n+1)n+1$	$\frac{2n^3-n^2}{3n^2} \approx \frac{2n}{3}$
	$\frac{2n^3-n^2}{3n^2} \approx \frac{2n}{3}$

Table 3 Computational intensities for the outer product method.

The computational intensities of these three matrix multiplication algorithms are compared in Fig. 2. The comparison is based on the matrix size 1000×1000, i.e., $n = 1000$. As seen in this figure, the inner product algorithm yields the highest computational intensity among the three, followed by the middle product method, and the outer product algorithm is the lowest. The reason is that the inner product method completes the sequence of computing one inner product before starting the next. This computation sequence involves at most one row of matrix A and one column of matrix B at a time. On the other hand, the outer product method provides the maximum degree of parallelism among the three methods. Since this method computes all n^2 vector product terms at the same time, it requires access to all elements in matrix A and B before any vector product is completed, thus yielding the lowest computational intensity for the three approaches (about half of that for the inner product method) when the storage size is less than n^2. Once the storage size is above n^2, the inner product method has no advantage. When the storage size is below n, the inner product method is also better than the middle product method. This is because the middle product method computes n vector product terms

at a time, thus requiring accesses to n operands at a time.

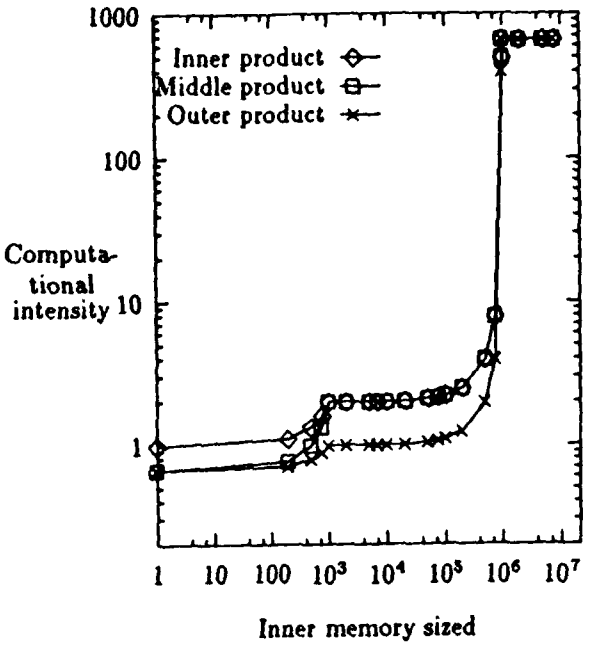

Figure 2. Computational intensity vs. inner memory sizes for matrix multiplication

2.2 Matrix Inversion

For inversion of an arbitrary $n \times n$ nonsingular matrix ($n > 2$) the pivot method [6] for which the pivot is determined by scanning a column for its maximum magnitude is assumed. Only one column is scanned for each pivot and the n pivots are determined in succession using the columns from left to right. The approximate size of the outer memory, M, must be at least $n(n+1)$. The number of flops, n_p, corresponding to each pivot is n comparisons, n divisions (or one reciprocation and n multiplications), $n(n-1)$ subtractions, and $n(n-1)$ multiplications so that $n_p = [2n + 2n(n-1)]\, n = 2n^3$

Computational intensities for several sizes of the inner memory are given in Table 4. Row interchanges have been ignored. As an example of computing the number of memory accesses, consider the third entry in the second column. For each column, the n column elements are input one at a time and the maximum element is determined. Then for each group of $m-1$ elements in the pivot row, the elements are input and divided by the pivot element. Then for each of the other $n-1$ rows, the element in the pivot column and $m-1$ other elements are input. After $m-1$ new values are computed, they are output. Finally, the $m-1$ new elements in the pivot row are output. Therefore, the number of accesses for each of the n columns is $n + [2(m-1) + (2m-1)(n-1)]\, \frac{n}{m-1} = \frac{2m-1}{m-1} n^2 + \frac{m-2}{m-1} n$ Also, the computational intensity as a function of m for $n = 1000$ is one of the curves in Fig. 3.

m	Computational intensity
0	$\frac{n}{2n+1} \approx \frac{1}{2}$
1	$\frac{2n}{3n+1} \approx \frac{2}{3}$
$1 < m \le n+1$	$\frac{2n(m-1)}{(2m-1)n+m-2} \approx \frac{2(m-1)}{2m-1}$
$n \bmod (m-1) = 0$	
$n+1 < m \le 2n$	$\frac{2n^2}{2n^2+n-m} \approx 1$
$m = (k+1)n$	
$k = 2, \ldots, n-1$	$\frac{n^2}{n^2-nk+k} \approx 1$
$m \ge n(n-1)$	n

Table 4 Computational intensities for matrix inversion.

2.3 Partial differential Equations

For examining the solution of partial differential equations, two-dimensional equations are assumed and the nearest neighbor approach is used [6], [2]. Suppose that the area of the solution is represented by an $n \times p$ array surrounded by $2(n+p)$ boundary points that are known. If the dependent variable is v and f is a known function, then there are constants a, b, c, d, and e such that $v_{ij} = a\, v_{i-1,j} + b\, v_{i+1,j} + c\, v_{i,j-1} + d\, v_{i,j+1} + e\, f_{ij} \quad i = 1, \ldots, p,\ j = 1, \ldots, n$

The solution is found by q iterations over the entire array by incrementing i and j and using the new values of v_{ij} as they are computed. So that the outer memory can contain the constants and all values of v_{ij} and f_{ij}, the size of M must be at least $2(np + n + p) + 5$. First np multiplications are required to replace all f_{ij}s with $e f_{ij}$ and then, for each iteration, four multiplications and four additions are required for each point. Therefore, $n_p = (8q+1)\, n\, p \approx 8\, npq$.

Table 5 gives the computational intensities for several values of m and one of the curves in Fig. 3 gives the computational intensity as a function of m for $n = p = 1000$ and $q = 20$. For example, the fifth entry in the second column of Table 5 was found for $k = 0$ assuming the five weighting constants are input and then pn values of f_{ij} are input and pn values of $e f_{ij}$ are output. Then for each iteration the $e f_{ij}$ products, $2(n+p)$ boundary points, and old values of the np interior points are input, and the new values of the np interior points are output. For $k > 0$ there are k points that need to be input and output only once instead of q times.

2.4 Linear equations

Simultaneous linear equations can be solved efficiently by the Gaussian elimination method [6]. In the first phase of the Gaussian elimination, n simultaneous linear equations with n variables are reduced to

$$\begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= C'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= C'_2 \\ &\cdots \\ a'_{nn}x_n &= C'_n \end{aligned} \qquad (1)$$

In the second phase, variables x_1 to x_n can be obtained by the following computations:

$$x_n = \frac{C'_n}{a'_{nn}}$$
$$x_k = \frac{C'_k - \sum_{i=k+1}^{n} a'_{ki}x_i}{a'_{kk}}, \quad k = n-1, \ldots, 1 \quad (2)$$

The total number of computations in phase 1 is

$$3[n(n-1) + (n-1)(n-2) + \cdots + 2] = 3\sum_{i=1}^{n-1} i(i+1)$$
$$= \frac{1}{2}n(n-1)(2n-1) + \frac{3}{2}n(n-1) = n^3 - n.$$

The number of computations in Eq. (2) is

$$1 + 3 + \cdots + 2n - 1 = \sum_{i=1}^{n}(2i-1) = n^2. \quad (3)$$

Therefore, the total number of computations for solving n simultaneous linear equations is

$$C = n^3 + n^2 - n \quad (4)$$

The number of accesses to outer memory and corresponding computational intensities are given in Table 6. Also, see Fig. 3 for the computational intensity as a function of m for $n = 1000$. The number of memory accesses for reducing n simultaneous linear equations to Eq. (1) in the case of $m = 0$ is derived as follows. Since there is no internal memory in the processing units, all intermediate results must be stored back to memory and read in for later computations. The number of memory accesses is

$$7[n(n-1) + (n-1)(n-2) + \cdots + 2] = 7\sum_{i=1}^{n-1} i(i+1)$$
$$= \frac{7}{6}n(n-1)(2n-1) + \frac{7}{2}n(n-1) = \frac{7}{3}n(n^2-1) \quad (5)$$

And the number of memory accesses in Eq. (2) is

$$3n + \sum_{i=3}^{n+1} i + 3\sum_{i=1}^{n-1} i = 3n + (n-1)(n+2)$$
$$= n^2 + 4n - 2. \quad (6)$$

Summing Eqs. (5) and (6), the total number of memory accesses is

$$\frac{7}{3}n^3 + n^2 + \frac{5}{3}n - 2. \quad (7)$$

For the case of $m = 1$, the intermediate product in the innermost loop can be stored in the inner memory, thus eliminating two outer memory accesses, e.g., one write and one read of the intermediate product. So, the computational intensity is 7/5 times higher than the case when $m = 0$. In the case of $m = 3$, one can store the most often used coefficients a_{kk} and a_{ik} in the inner memory to further reduce the outer memory accesses to three accesses per loop. When more inner memory is available, one can store all the coefficients in the inner memory, thus the minimal number of outer memory accesses will be equal to $n(n+1)$. Since the total number of computations is $O(n^3)$, the computational intensity approaches n when $m = n(n+1)$.

m	Computational intensity
0	$\frac{8q+1}{11q+3} \approx \frac{8}{11}$
$1 < m \leq 4$	$\frac{(8q+1)np}{(11-m)npq+m+2np+1} \approx \frac{8}{11-m}$
6	$\frac{(8q+1)np}{(5np+2)q+2np+5} \approx \frac{8}{5}$
$m = p + 6$	$\frac{(8q+1)np}{(4np+n+p+1)q+2np+5} \approx 2$
$m = 2p + 6 + k$, $k = 0, \ldots, n(p-1)$	$\frac{(8q+1)np}{(3np+2n+2p)q+2np+5-2k(q-1)} \approx \frac{8}{3}$
$m = 2p + 6 + np + k$, $k = 0, \ldots, n(p-1)$	$\frac{(8q+1)np}{(np+2n)q+2p+4np+5-k(q-1)} \approx 8$
$m = 2p + 6 + 2np + k$, $k = 0, \ldots, 2n-1$	$\frac{(8q+1)np}{3np+2nq+2p+5-k(q-1)} \approx \frac{8}{3}q$
$m \geq 2(np+n+p)+5$	$\frac{(8q+1)np}{3np+2n+2p+5} \approx \frac{8}{3}q$

Table 5 Computational intensities for partial differential equation solution.

m	Computational intensity
0	$\frac{n^3+n^2-n}{\frac{7}{6}n^3+n^2+\frac{5}{3}n-2} \approx \frac{3}{7}$
1	$\frac{n^3+n^2-n}{\frac{7}{6}n(n-1)(2n-1)+(\frac{7}{2}n+1)(n-1)+3} \approx \frac{3}{5}$
3	$\frac{n^3+n^2-n}{\frac{7}{6}n(n-1)(2n-1)+(\frac{7}{2}n+2)(n-1)} \approx 1$
$n+2$	$\frac{n^3+n^2-n}{\frac{7}{6}n(n+1)(2n+1)+n(n+3)} \approx 1.5$
$2n$	$\frac{n^3+n^2-n}{\frac{7}{6}n(n-1)(2n-1)+\frac{7}{2}(n+5)} \approx 1.5$
n^2+n	$\frac{n^3+n^2-n}{n^3+2n} \approx n$

Table 6 Computational intensities for solving linear equations using Gaussian elimination.

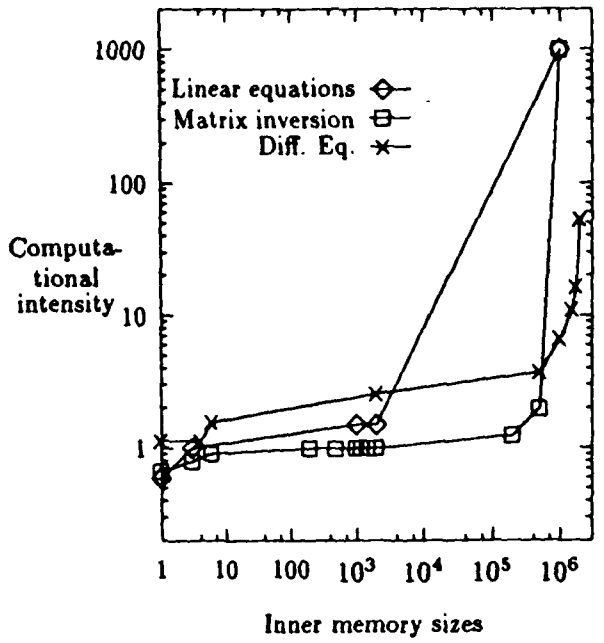

Figure 3. Computational intensity vs. inner memory sizes for linear equations, matrix inversion, and partial differential equations.

3 Time Analysis

One use of computational intensity is in time analysis for which, at any level i in the hierarchy, the memory communicates with only the memories at the $i-1$ and $i+1$ levels. Consider a hierarchy of $r+1$ levels. If

- m_i = number of locations in the ith memory level, $i = 0, 1, \ldots, r$, where the 0th level includes only processing (i.e., $m_0 = 0$).
- n_i = number of operand accesses from the $(i+1)$th level to the ith level, $i = 0, 1, \ldots, r-1$.
- s_p = processor speed in megaflops per second.
- s_i = memory access speed in megaoperands per second of level i and includes the miss determination time associated with level $i-1$, $i = 1, \ldots, r$.
- $I(m_{i-1})$ = computational intensity at the boundary between levels $i-1$ and i, $i = 1, \ldots, r$, as a function of the size of memory inside that boundary.

then the time required to complete an algorithm, assuming all memory levels except the outer level operate like ordinary caches, is

$$T_e = \frac{n_p}{s_p} + \sum_{i=1}^{r} \frac{n_i}{s_i}$$

Let T_n be the normalized quantity

$$T_n = \frac{s_p}{n_p} T_e = 1 + \sum_{i=1}^{r} \frac{R_i}{I_{i-1}} \qquad (8)$$

where $R_i = s_p/s_i$ and $I_{i-1} = \frac{n_p}{n_i} = I(m_{i-1}), i = 1, \ldots, r$.

Now suppose that instead of using ordinary cache at level j, it is assumed that the memory at this level can prefetch operands from the outer levels and automatically supply them to the inner levels. Then there can be overlapping of processing and memory accessing in the inner levels with the memory accessing in the outer levels. This overlapping implies that the summation in Eq. (8) can be replaced with

$$T_n = \begin{cases} max\{1, \sum_{i=1}^{r} \frac{R_i}{I_{i-1}}\} & j = 1 \\ max\{1 + \sum_{i=1}^{j-1} \frac{R_i}{I_{i-1}}, \sum_{i=j}^{r} \frac{R_i}{I_{i-1}}\} & j = 2, \ldots, r \end{cases}$$

In the extreme, for which prefetching is done at all levels so that overlapping is maximized

$$T_n = max\{1, \frac{R_1}{I_0}, \ldots, \frac{R_r}{I_{r-1}}\}$$

This is minimized when $\frac{R_i}{I_{i-1}} \leq 1$ or $s_p \leq s_i I_{i-1}$, $i = 1, \ldots, r$ For the slowest speeds, $s_i = s_p/I_{i-1}, i = 1, \ldots, r$.

Next let us assume no prefetching, but that the $(j-1)$th level can access the $(j+1)$th level directly. For input, operands are stored in the $(j-1)$th and jth levels simultaneously and for output, operands are stored in the jth and $j+1$th levels simultaneously. Then

$$T_e = \frac{n_p}{s_p} + \sum_{i=1}^{r} \frac{n_i}{s_i} - \frac{n_{j+1}}{s_j} \quad \text{and} \quad T_n = 1 + \sum_{i=1}^{r} \frac{R_i}{I_{i-1}} - \frac{R_j}{I_j}$$

In the extreme for which all memories can access the processor level at the same time as the other memory levels and there is prefetching and direct accessing at all levels

$$T_n = \max_i \{1, R_i \left[\frac{1}{I_{i-1}} - \frac{1}{I_i}\right], \frac{R_r}{I_{r-1}}\}$$

The minimum of T_n for the slowest memory speeds occurs when

$$s_i \approx s_p \left[\frac{1}{I_{i-1}} - \frac{1}{I_i}\right], \quad i = 1, \ldots, r-1, \quad \text{and} \quad s_r = \frac{s_p}{I_{r-1}}$$

4 Cost Minimization

Another use of the computational intensity is in the minimization of the cost of a processing system that includes a memory hierarchy of $r+1$ levels. The total cost is

$$c = c_p(s_p) + \sum_{i=1}^{r} c_m(s_i, m_i), \quad m_r = M$$

where $c_p(s_p)$ is the cost of the processing logic as a function of processing speed and c_m is the cost of memory as a function of memory access speed in operands/s and size in operand locations. Now suppose that c is to be minimized relative to the constraints that a given algorithm must be performed within a specified time T, $s_p > 0, s_i > 0$ for all i and $m_i > 0$ for all i. The time constraint normalized

by n_p, assuming no overlapping of processing and memory accessing, is

$$T \geq \frac{T_1}{n_p} = \frac{1}{s_p} + \sum_{i=1}^{r} \frac{1}{s_i I_{i-1}}. \qquad (9)$$

For a realistic system, c_p and c_m are strictly monotonic increasing positive functions (MIPFs) that approach infinity as speed approaches infinity. Also c_m is a strictly MIPF that approaches infinity as memory size approaches infinity. Therefore, the minimum c must occur when the inequality becomes an equality and

$$s_p = \frac{1}{T - \sum_{i=1}^{r} \frac{1}{s_i I_{i-1}}} > 0 \qquad (10)$$

But, s_p is a strictly monotonic decreasing positive function (MDPF) of each s_i and, hence, $c_p(s_p)$ is a strictly MDPF of each s_i. This implies that, for each s_i, c is the sum of a strictly MDPF and a strictly MIPF. Therefore, c has a minimum relative to the s_is that occurs at exactly one point $(s_1', \ldots, s_r')$. Similarly, because $I(m)$ is a MIPF of m, c is known to have a minimum relative to the m_is. Moreover, because $I(m)$ may not be strictly monotonic increasing, the minimum may occur at several points. But there is at least one point $(s_1', \ldots, s_r', m_1', \ldots, m_{r-1}')$ at which c is a minimum. The speed s_p can be determined from Eq. (10) and $m_r = M$ as given.

As an example, let us assume that

$$c = Cs_p + \sum_{i=1}^{r} f(T)\, s_i (Am_i + B), \qquad (11)$$

where A, B, and C are constants and

$$f(T) = \begin{cases} 10 & T < 10ns \\ 1 & T \geq 10ns \end{cases},$$

and minimize the cost of the memory hierarchy relative to the inner product algorithm for multiplying 1000 × 1000 matrices. The factor $f(T)$ is to compensate for a change to a very fast technology. Suppose that $C = 5 \times 10^{-4}$ dollars per flops/s, $A = 10^{-11}$ dollars per flops/s per operand, $B = 10^{-6}$ dollars per flops/s, $M = 1.6 \times 10^7$ operands, 1, 200, 500, 1001, 2001, 3001, 4001, 251001, 501001, 751001, and 10001000 operands are the possible memory sizes, and 10, 15, 20, 25, 30, 50, 100, 200 and 400 M operands/s are the possible memory speeds.. Then the minimum total cost as a function of actual processing speed (as opposed to the speed of the processor) is as shown in Fig. 4. This figure has three curves and each cure corresponds to a number of levels. A four-level hierarchy was also considered, but in no case did the fourth level permit a reduction in the total cost.

The above has assumed nonoverlapping of processing and memory accessing. If we now assume that the memory at level j can prefetch operands from the outer levels and automatically supply them to the inner levels, then there can be overlapping of processing and the memory accessing of the inner levels with the memory accessing of the outer levels. This overlapping implies that the summation in inequality Eq. (9) can be replaced with the maximum of two summations and

$$T \geq \begin{cases} max\{\frac{1}{s_p}, \sum_{i=1}^{r} \frac{1}{s_i I_{i-1}}\} & j = 1 \\ max\{\frac{1}{s_p} + \sum_{i=1}^{j-1} \frac{1}{s_i I_{i-1}}, \sum_{i=j}^{r} \frac{1}{s_i I_{i-1}}\} & j = 2, \ldots, r \end{cases}$$

For $j = 2, \ldots, r-1$, assume that

$$\frac{1}{s_p} + \sum_{i=1}^{j-1} \frac{1}{s_i I_{i-1}} < \sum_{i=j}^{r} \frac{1}{s_i I_{i-1}} \qquad (12)$$

when the cost c is at its minimum. But, without increasing T, less memory could be used to construct the inner memories and the cost could be reduced, thus contradicting the original assumption. Similarly, if inequality (12) is reversed the cost could be reduced by reducing the amount of outer memory without increasing T. Therefore, the minimum c occurs when inequality (12) is replaced by an equality and both sides are equal to T. Similar arguments could be made for the cases $j = 1$ and $j = r$. This implies that the minimum occurs when

$$T = \begin{cases} \frac{1}{s_p} = \sum_{i=1}^{r} \frac{1}{s_i I_{i-1}} & j = 1 \\ \frac{1}{s_p} + \sum_{i=1}^{j-1} \frac{1}{s_i I_{i-1}} = \sum_{i=j}^{r} \frac{1}{s_i I_{i-1}} & j = 2, \ldots, r \end{cases}$$

Therefore, two of the speed variables, say s_p and s_r, can be expressed in terms of T and eliminated from the minimization process.

In the extreme, complete overlapping for which all memories operate independently in supplying and storing the needed operands may be assumed. In this case the minimum occurs when

$$T = \frac{1}{s_p} = \frac{1}{s_1 I_0} = \cdots = \frac{1}{s_r I_{r-1}}$$

and all of the speed variables can be expressed in terms of the T and the computational intensities, and the expression to be minimized becomes

$$c = c_p(\frac{1}{T}) + \sum_{i=1}^{r} c_m\left(\frac{1}{T I_{i-1}}, m_i\right), \quad m_r = M$$

Because $c_p(1/T)$ is a constant for a given value of T, it may be eliminated from the minimization process. If it is assumed that $c_m = D\,s(m+B)$, $D > 0$, then the expression to be minimized is

$$\sum_{i=1}^{r} \frac{Am_i + B}{I_{i-1}}, \quad m_r = M$$

and the minimal cost is

$$c = c_p(\frac{1}{T}) + \frac{D}{T} \sum_{i=1}^{r} \frac{Am_i + B}{I_{i-1}}, \quad m_r = M. \qquad (13)$$

If overlapping and the same cost function with the same parameters used to generate Fig. 4 are assumed, then the minimum total cost as a function of the number of levels is shown in Fig. 5 for four problem sizes. The memory sizes were limited as before, but the memory speeds were not limted. Note that from Eq. (11) the total cost increases with the processing speed, which was assumed to be 100 Mflops/s while generating Fig. 5. Also, note that the total cost depends very little on problem size and there is no gain in using four memory levels.

5 Summary and Conclusions

In this paper, the computational intensity for various algorithms are investigated. Computational intensity directly affects the hit ratios in a hierarchical memory system and is, therefore, a major factor on memory performance.

This paper also develops expressions showing the relationship between the computational intensity and the speed and cost of hierarchical memory systems. These expressions can be used as design tools in determining the optimal size and speed for each memory level without relying on time-consuming simulations.

A cost minimization example is presented which analyzes a hierarchy for both the overlapped and nonoverlapped cases. Although the data resulting from only one algorithm and one set of cost parameters is shown, a variety of algorithms and cost parameters were examined and it was noted that in none of trial minimizations did four memory levels show a significant cost improvement over three memory levels. However, for the nonoverlapped trials the memory speeds were limited to 10 M operands/s and above.

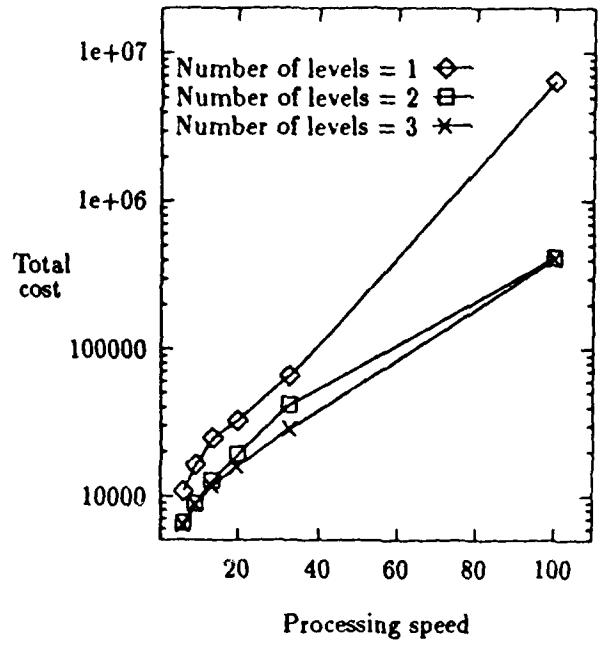

Figure 4. Total cost vs. processing speed for various numbers of memory levels.

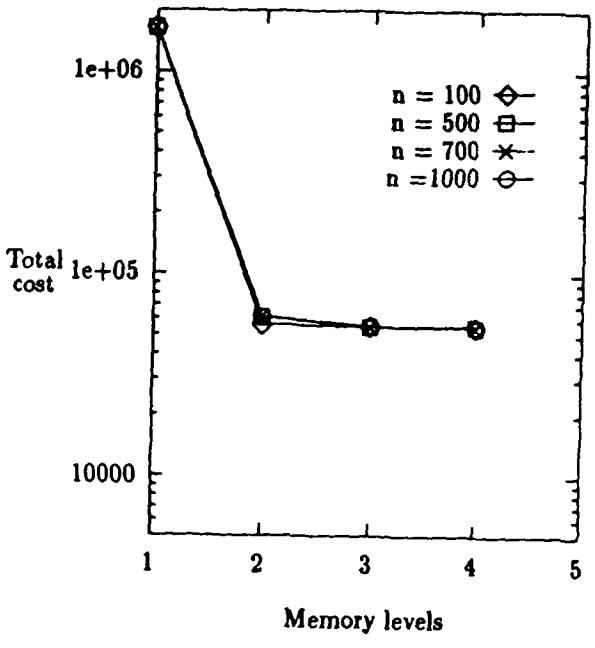

Figure 5. Total cost vs. memory levels with overlapping.

References

[1] S. S. Gurindar and W.-C. Hsu, "The use of intermediate memories for low-latency memory access in supercomputer scalar units," *Journal of Supercomputing*, no. 4, pp. 5–21, 1990.

[2] H. S. Stone, *High-performance Computer architecture*, 3rd Edition Addison-Wesley, 1993.

[3] K. Olukotun, T. Mudge, and R. Brown, "Performance optimization of pipelines primary caches," *Proc. the 19th Annual International Symposium on Computer Architecture Conference*, pp. 181–190, May 1992.

[4] J. H. Tang and E. S. Davidson, "An Evaluation of Cray I and Cray X-MP Performance on Vectorizable Livermore FORTRAN Kernels," *Proc. 1984 Int'l Conf. on Supercomputing*, pp. 510–518, 1988.

[5] R. Hockney and C. Jesshope, *Parallel Computers 2*, Adam Hilger: Bristol, England, 1988.

[6] B. Carnahan, H. Luther, and J. Wilkes, *Applied Numerical Methods*, New York: John Wiley and Sons, Inc., 1969.

ATTACHMENT D

A MODULE–SLICED HIGH YIELD WSI MEMORY SYSTEM [1]

Yi-Chieh Chang and Saji Jorge

Electrical Engineering Department
The University of Texas at El Paso
El Paso, TX. 79968-0523
(915) 747-6960
chieh@ee.ep.utexas.edu

Jung Hwan Kim

Center for Advanced Computer Studies
Univ. of Southwestern Louisiana
Lafayatte, LA 70504-4330
(318) 231-6732
kim@cacs.usl.edu

Abstract

Low yield is one of the practical difficulties in the design of WSI systems, such as array processors or WSI memories. The conventional row–column memory cells organization is not suitable for WSI memory systems due to the long signal delay on a wafer and a much more complicate procedure for replacing a defect row or column of memory cell. To alleviate these difficulties, a module–sliced WSI memory system is proposed for high yield WSI memory systems. The basic unit of the WSI memory system is a module which consists of a memory bank, a module comparator, a module register, and a row-column decoder. The WSI memory system is organized in a two level row/column structure. The first level is a two dimensional mesh with the basic unit of a module. Within each module, i.e. the second level, the memory bank is organized in a conventional rows and columns of memory cells.

The most important feature of the proposed WSI memory system is that the reconfiguration of the faulty memory system into a fault-free memory system is done straightforward without employing any reconfiguration algorithm by using the module address in each module. Each module in the WSI memory system is a complete memory system and its operation is independent of any other module. An effective module address stored in the module register can activate a module if its fault-free, otherwise a dummy address can be stored in the module register to bypass a faulty module without the needs for reconfiguration. Since the module comparator, the register, and the row-column decoder inside a module are the extra hardware required for the module-sliced WSI memory system which will in turn decrease the yield of the memory module, we found through simulation the optimal module size which will maximize the yield on the WSI memory system. Our studies showed that for a 64 Mb and 256 Mb memory system, the optimal module size is 16 Kb, while the optimal module size is 64 Kb for a 1024 Mb memory system. The yield rate of the WSI memory system can be as high as 70% for the 64 Mb memory system and 40% for the 1024 Mb memory system using a 0.6 μm technology with a defect density of four defects per square centimeter.

Index Terms—WSI memory system, module-sliced, yield rate, reconfiguration, defect.

[1]The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-93-1-1343. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agency.

1 Introduction

Due to the advances on VLSI technologies and studies in the reconfigurable fault-tolerant architectures in the past decades, many high-yield VLSI systems built on an entire wafer has been reported recently [1, 2, 3, 4]. Especially, a 3-D WSI signal processing system using indium bumps for wafer-to-wafer interconnections has been reported to be able to stack multi-wafers in a high density VLSI system [3, 5]. In such a system, a WSI memory system is needed for storing the image data or supporting the processing elements on another wafer through the vertical wafer-to-wafer interconnections.

The conventional row–column memory cells organization is not suitable for WSI memory systems due to the long signal delay on a wafer and a much more complicate procedure for replacing a defect row or column of memory cell. Moreover, since the yield of a module decreases with the increase of its size, the yield of a system can be improved by decomposing a large system into several smaller submodules, or using the module-sliced approach. The submodules will collectively perform the function of the original large module. In an earlier paper [6], we have proposed a reconfigurable fault-tolerant segmented array processor (RFTSAP) structure to realize the module-sliced approach for high yield WSI array processors. In this paper, our fault-tolerant WSI memory system can be reconfigured without employing any reconfiguration algorithm by using a module address in each module.

In this paper, a module–sliced memory architecture is proposed for high yield WSI memory systems. The basic unit of the WSI memory system is a module which consists of a memory bank, a module comparator (MC), a module register (MR), and a row-column decoder. The WSI memory system is organized in a two level row/column structure. The first level is a two dimensional mesh with the basic unit of a module. Within each module, i.e. the second level, the memory bank is organized in conventional rows and columns of memory cells. The size of the memory bank is a multiple of 4, such as 16 Kb, or 64 Kb, organized in rows and columns. The actual number of rows and columns depends on the size of the memory bank.

Each module in the WSI memory system is a complete memory system, and its operation is

independent of any other module. Addressing a memory cell in the module-sliced WSI memory system is done in two steps which can be processed concurrently. A module address is sent to the MC to select a module, and a cell address is sent to every module to select one memory cell within each memory module. If the memory bank in a module has one or more defective cells, a dummy module address which will permanently disable the faulty memory module will be stored in the MR. On the other hand, if a memory module is tested as fault-free, an effective module address will be loaded to the MR which will activate the memory module when the MC detects a match between the stored module address and the address on the address bus. However, if the MC or MR, or the address and control signal line in the memory module is faulty, an entire column of memory modules will be discarded since the defective memory module may affect the read/write operations of other fault-free memory modules in the same column; this is considered as a *serious defect* in the module-sliced WSI memory system.

Since a memory module will be disabled if there is more than one defective cell in the module, a smaller memory bank in a module will generally minimize the percentage of wasted memory cells. However, since the MC, MR, and the row-column decoder inside a module are the extra hardware required for the module-sliced WSI memory system, this will in turn decrease the yield of the memory module. Moreover, the probability of having a serious defect on a memory module depends on the relative area ratio of the extra hardware to the memory bank. A smaller memory module will have a higher probability of having a serious defective which will destroy an entire column of memory modules. Thus, it is desirable to derive the optimal module size which will maximize the yield on the WSI memory system.

Our analysis showed that for a 64 Mb and 256 Mb memory system, the optimal module size is 16 Kb, while the optimal module size is 64 Kb for a 1024 Mb memory system. The yield rate of the WSI memory system can be as high as 70% for the 64 Mb memory system and 40% for the 1024 Mb memory system using a 0.6 μm CMOS technology with a defect density of four defects per square centimeter.

The rest of this paper is organized as follows. The architecture, addressing technique,

and the procedure to bypass the faulty modules in the module-sliced WSI memory system is described in Section 2. The analysis and derivation of the optimal module size based on a 6-transistor SRAM layout is discussed in Section 3. This paper concludes with Section 4.

2 Architecture and operation of the module-sliced memory system

The basic unit of the module-sliced WSI memory system is a module which consists of a memory bank, a MC, a MR, and a row-column decoder. Assume the total memory capacity of a memory system is 2^N. The WSI memory system is organized in a two level row/column structure. The first level is a two dimensional mesh with a total number of 2^m modules organized in a $2^{m/2} \times 2^{m/2}$ (assume m is an even number) square mesh. A module decoding circuitry in the first level is used to select one module out of the 2^m modules to load an effective module address to the MR in each module. In each module, the size of the memory bank is 2^ℓ where $m + \ell = N$. The memory cells in the memory bank is organized in a conventional row-column fashion, i.e., $2^{\ell/2} \times 2^{\ell/2}$ (assume ℓ is also an even number). The MR stores m bits module address, the MR has two operation mode: configuration mode and operation mode. In the configuration mode, the most significant m bits on the address bus will be stored in the MR. In the operation mode, the content of the MR will not be changed. The MC compares m bits address in parallel between the MR (operation mode) and the most significant m bits on the address bus.

The row-column decoder decodes the least significant ℓ bits on the address bus and selects one cell from the memory bank. The block diagram of a memory module is shown in Fig. 1 and the organization of a 4×4 modules is shown in Fig. 2.

2.1 Addressing procedure in the module-sliced WSI memory systems

To address one memory cell from the entire memory system, two steps addressing process should be carried. The most significant m bits address lines will be sent to the MC in each module The MC compares the m bits address stored in the MR with the address lines. If there is a match, the memory module will be activated, otherwise the entire module will be disabled. At the same time, the row-column decoder in each module will receive the least significant ℓ bits address and an unique cell will be selected from the memory bank. If every fault-free memory module has an unique module address, only one module will be activated from the most significant m bits address, thus only one memory cell will be accessed at a time. One of the advantage of the module-sliced memory system is that the MC and the row-column decoder in each module can operate at the same time, thus reducing the address decoding time significantly.

2.2 Bypassing faulty memory modules

Due to the imperfect manufacturing procedure, it is almost impossible to fabricate a WSI memory system of the size more than 64 Mb without any defects. If no spare rows or columns of memory cells are provided in each memory module, the module needs to be bypassed or permanently disabled if there is only one defect in the memory bank. Although many reconfiguration algorithms have been reported to be able to reconfigure the WSI systems in the presence of defects [7, 8, 9], the defective modules in the module-sliced WSI memory system can be bypassed without using any sophisticated reconfiguration algorithm. Bypassing a faulty module can be easily done by storing a dummy module address in the MR so that the module will never be activated whenever an effective address is inserted.

However, if the MR or the MC are defective, it is considered as a serious defect since it is impossible to disable the module by storing a dummy address to the MR. Moreover, if the address line or the control signal line in the module is defective, it is also impossible to bypass the defective module. In such a case, an entire column of modules must be discarded

by disconnecting the power connection to that column. In order to reduce the number of good memory modules to be discarded due to the serious defect, a complete set of address bus and control signal lines is passed to each column of memory modules, so that any serious defect on one column of memory modules will only affect that column, not the adjacent columns. Note that the probability of having a serious defect in a memory module depends on the ratio of the areas for the memory bank and the extra hardware, such as the MC, MR, and the signal lines.

The configuration of a 2 × 4 array out of a 4 × 4 array is shown in Fig. 3.

2.3 Spare rows and columns in the memory bank

When the size of the memory bank is large, for instance, larger than 64 Kb, it is found that most of the defects will fall on the memory bank because the hardware overhead of the MC, MR, and control lines is almost negligible. Thus, it is desirable to provide some spare rows and columns in the memory bank, so that the memory module can be repaired by replacing the defective rows or columns from the spares. The improvement of the yield rate by providing spare rows and columns will be discussed in the following section.

3 Yield rate analysis

The yield rate for a VLSI system is usually defined as the ratio between the number of good chips and the total number of chips fabricated on a wafer. However, for the WSI memory system, the entire system is built on a wafer, the yield rate definition for the VLSI system needs to be modified for describing the yield rate of the WSI memory system. Note that each memory module in the proposed module-sliced memory system is a complete system and every fault-free memory module can be configured into a functioning memory system with a reduced usable memory capacity. Since a memory module will be either activated if it's fault-free, or discarded if it's faulty, the ratio of the number of fault-free modules to the total number of modules fabricated on the wafer is equivalent to the yield rate definition for a VLSI system.

Let the number of faulty memory modules be f, the yield rate of the WSI memory system is $\frac{2^m - f}{2^m}$. A memory module is considered as faulty when there is more than one defect in the memory bank or an entire column of memory modules will be considered as faulty if any one of the memory modules in the same column has a serious defect. Assuming the defect density D is a constant, such as 4 defects per square centimeter, the total number of defects on an entire wafer is bounded by the product of $D \times A$ where A is the area of the wafer. Thus, the smaller the module size (smaller ℓ), the yield rate will be higher since $2^m = 2^N/2^\ell$ will increase by reducing ℓ. However, the relative area ratio for the MC, MR, and signal lines to the memory bank will increase when the size of the memory bank decreases, thus increasing the probability of having a serious defect.

In order to find the optimal module size, we consider the layout of a 6-transistor SRAM as an example in this study. A complete VLSI layout of a 2 x 2 memory module with 64 bits memory bank is shown in Fig. 4. From this figure, it is easy to see that when the memory bank size is very small, the hardware overhead is almost 40% of the entire memory module, thus dramatically increasing the probability of having a serious defect. Based on the layout in Fig. 4, we have calculated the module size for 4 Kb and 64 Kb in a 256 Mb memory system for 1.0 to 0.4 μm design technologies. The calculated areas are given in Tables 1 and 2.

3.1 Simulation results

Based on the areas calculated from the basic layout of the 64 bits memory module, we have randomly generated defects on the wafer assuming D = 2, 3, 4, and 6 for 1.0 μm, 0.8 μm, 0.6 μm, and 0.4 μm design technologies, respectively. The defects are uniformly distributed over the entire wafer. If a defect falls on a memory bank, the memory module is marked as faulty. If a serious defect is found, all memory modules in the same column are marked as faulty and cannot be repaired by the spare rows or columns in the memory bank.

Figs. 5 and 6 show the yield rate of the module-sliced WSI memory system vs. various module size without any spare rows or columns in the memory bank. It is found that the

optimal module size of 64 Mb and 256 Mb memory system is 16 Kb while the optimal module size is 64 Kb for the 1024 Mb memory system. With two spare rows or columns in the memory module (for module size larger than 64 Kb) as shown in Fig. 7, the optimal module size for 64 Mb and 256 Mb memory shifts to 64 Kb and the yield rate can be as high as 98% for the 64 Mb memory system. The optimal module size for the 1024 Mb memory shifts to 256 Kb in this case. As shown in Fig. 8 with 4 spare rows or columns, the optimal module size remains the same for all three memory systems. Note that the yield rate for the 1024 Mb system is as high as 75% with four spare rows or columns in the memory bank using the optimal module size (256 Kb). Several simulations were run to study the effect of increasing the spare rows and columns to the optimal module size and it is found that the optimal module size ranges between 64 Kb and 256 Kb with up to 32 spare rows or columns.

Furthermore, we found that when the module size is less than 16 Kb, the probability of having a serious defect is higher than 10%, thus, the yield rate will not be increased significantly even if spare rows or columns are provided in the memory bank since the serious defect cannot be repaired by the spare memory cells.

4 Conclusions

In this paper, we have studied a module-sliced WSI memory system which allows defective memory modules to be easily bypassed from the fault-free modules. Each memory module is a complete system and its operation is independent of any other modules. An optimal module size which maximizes the yield rate of the WSI memory system is derived from the simulation study.

Several advantages of the module-sliced WSI memory system are as follows. First, the reconfiguration of the faulty memory system into a fault-free memory system is done straightforward without employing any reconfiguration algorithm by using the module address in each module. Second, the address decoding is done in two steps in parallel, i.e. comparison in the MC and decoding in the row-column decoder, thus reducing the memory access time. Third, a

faulty module can be easily bypassed by storing a dummy address in the MR if the defects are in the memory bank without imposing any complicate reconfiguration algorithm. Fourth, since each module is a complete system it is possible to achieve parallel testing on each module, thus reducing the testing complexity from 2^N to 2^ℓ which is several order of magnitudes less than 2^N.

Several interesting topics worth to be further investigated in the future are (1) the modification of the memory bank so that the memory module can be read and write in a byte, word, or double word width, (2) Addition of a self-testing circuitry in each memory module so that parallel testing can be implemented, and (3) performing a timing analysis on the WSI memory system.

References

[1] V. K. Jain, H. Hikawa, and D. Keezer, "A WSI rapid prototyping architecture," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 35–44, 1992.

[2] S. Horiguchi and X. X. Zhang, "WSI architecture of FFT," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 35–54, 1992.

[3] S. Hedge, C. Habiger, and R. Lea, "3D-WASP devices for on-line signal and data processing," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 11–21, 1994.

[4] J. E. Brewer, L. G. Miller, I. H. Gilbert, J. F. Melia, and D. Garde, "A single-chip digital signal processing subsystem," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 265–272, 1994.

[5] R. Williams and O. Marsh, "Future WSI technology: Stacked monolithic WSI," *IEEE Trans. on Components, Hybrid, adn Manufacturing Technology*, vol. 16, no. 7, pp. 610–614, November 1993.

[6] Y. C. Chang and K. G. Shin, "A module-sliced approach for high yield WSI array processors," *IEEE ICCD*, pp. 500–503, 1989.

[7] W. Shi, W. K. Fuchs, and R. Mann, "Optimal spare allocation for defect-tolerant VLSI," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 193–199, 1992.

[8] D. D. Sharma, F. J. Meyer, and D. K. Pradhan, "Yield optimization of redundant multi-megabit RAM's using the center-satellite model," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 200–206, 1992.

[9] Y.-Y. Chen, Y.-S. Shyu, and C.-H. Cheng, "An effective framework for fault-tolerant VLSI/WSI arrays based on hybrid redundancy approach," *IEEE Int'l Conf. on Wafer Scale Integration*, pp. 153-162, 1994.

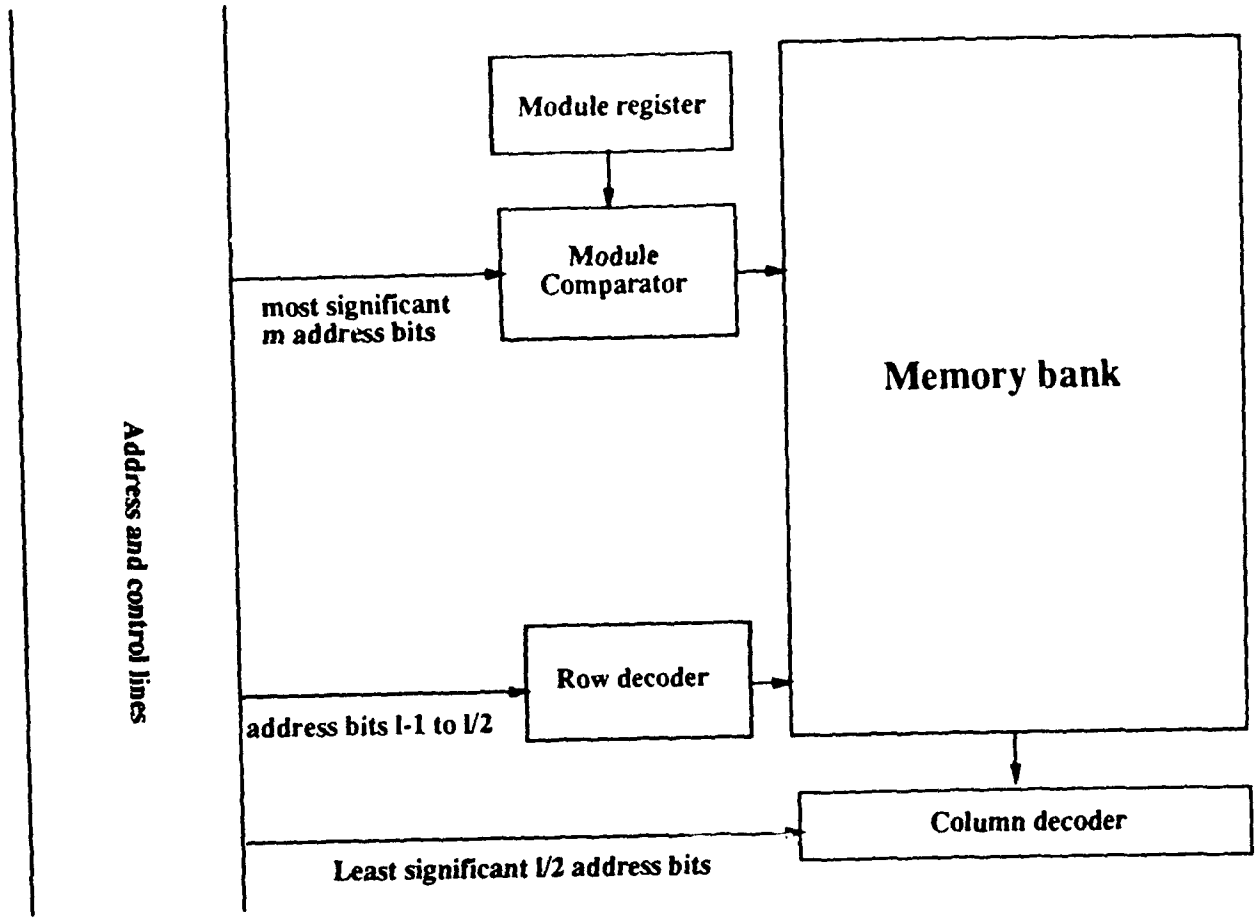

Fig. 1. Block diagram of a memory module

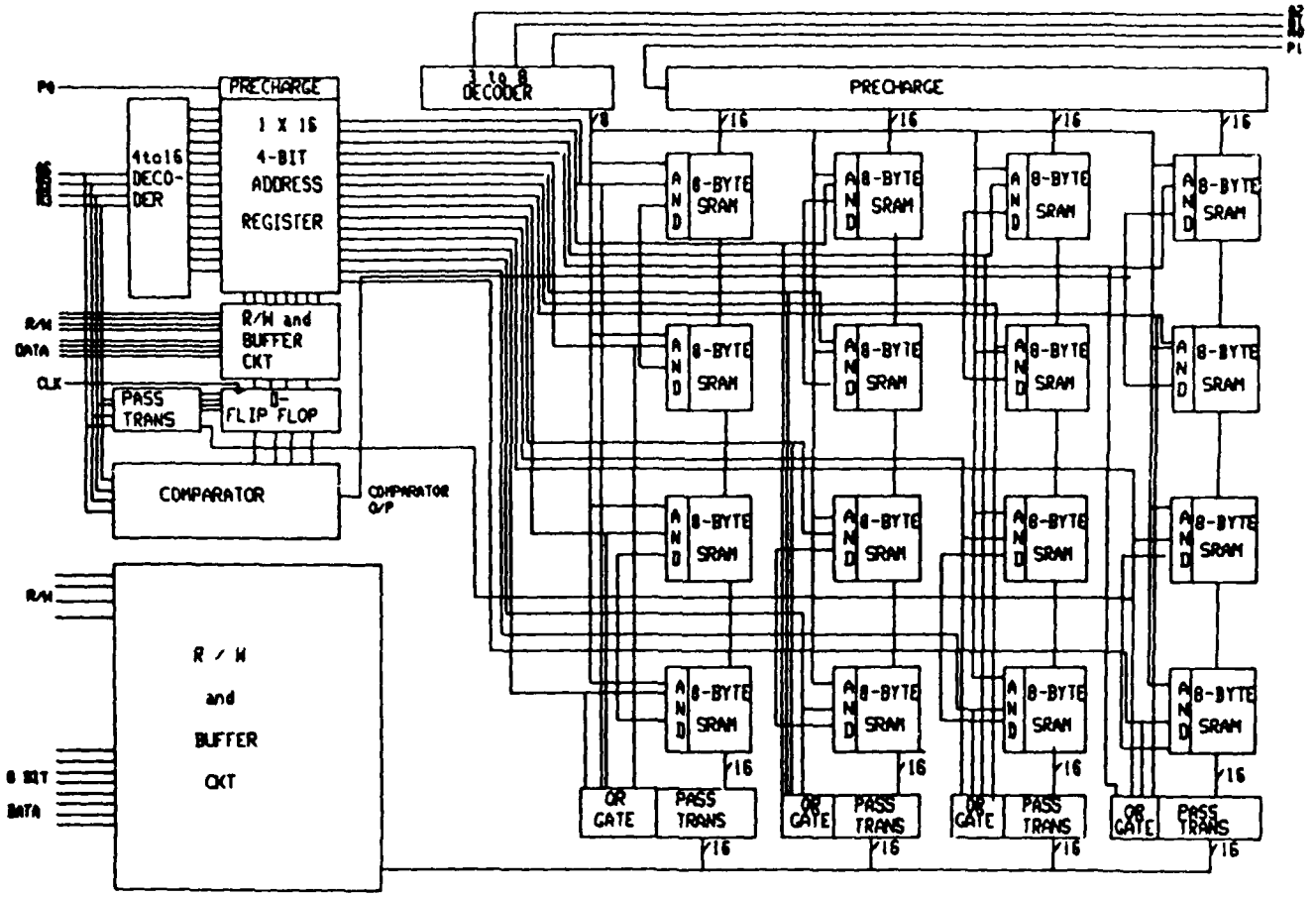

Fig. 2. Block diagram of a 4 x 4 module-sliced WSI memory system

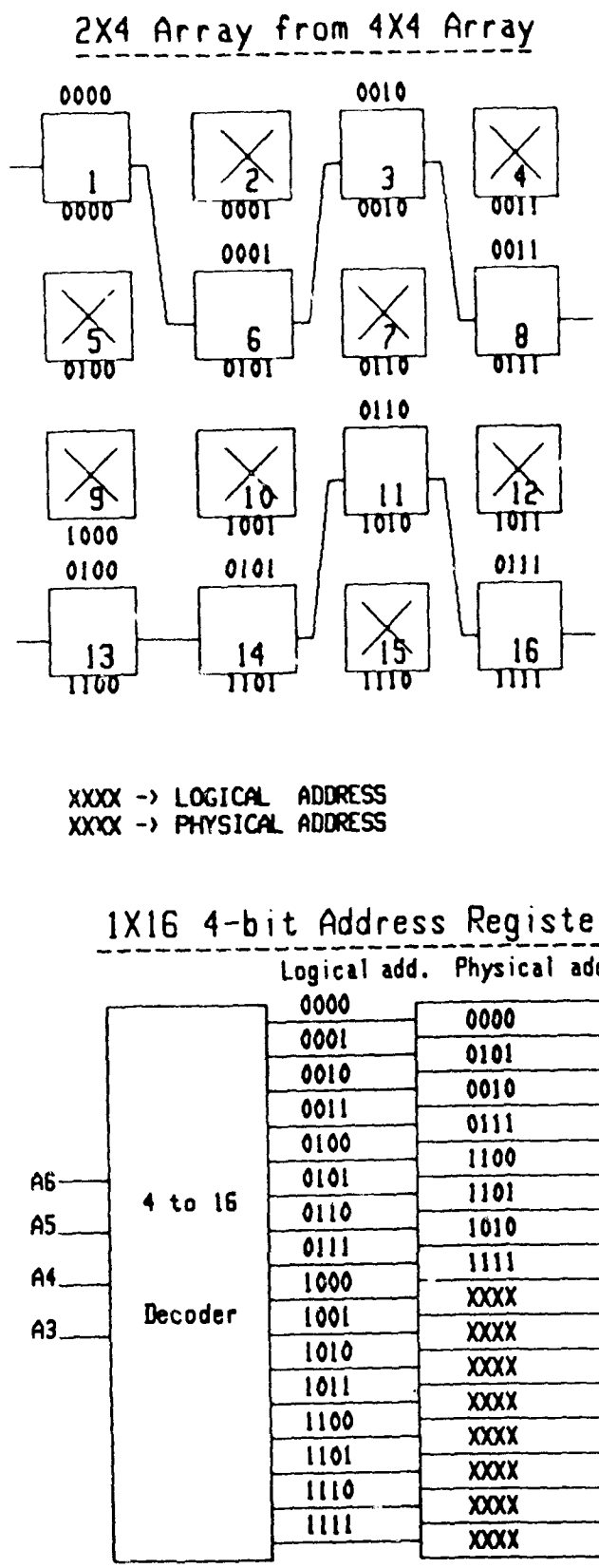

Fig. 3. Configuration of a 2 x 2 modules out of a 4 x 4 modules

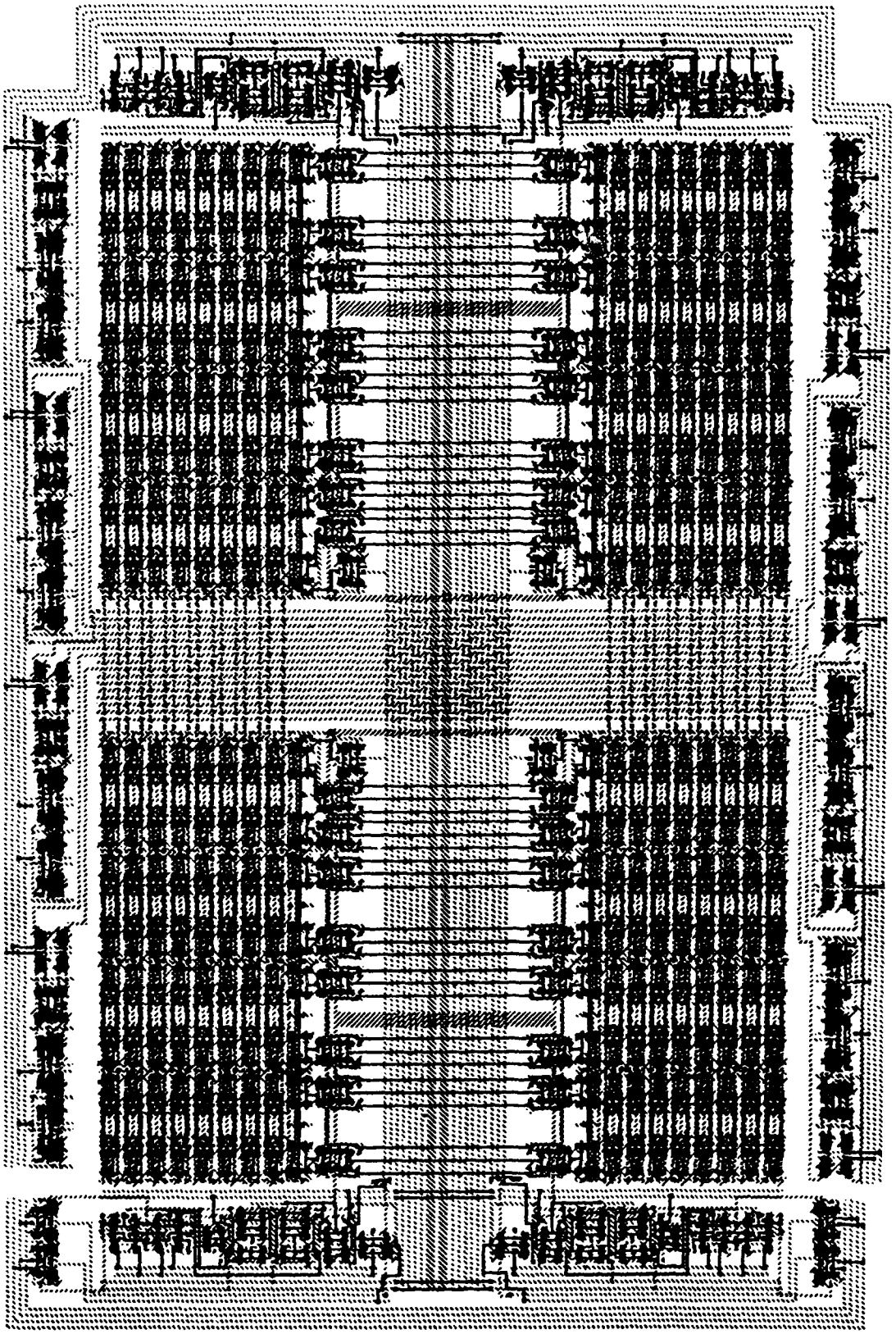

Fig. 4. VLSI layout of a 2 x 2 module-sliced WSI memory system

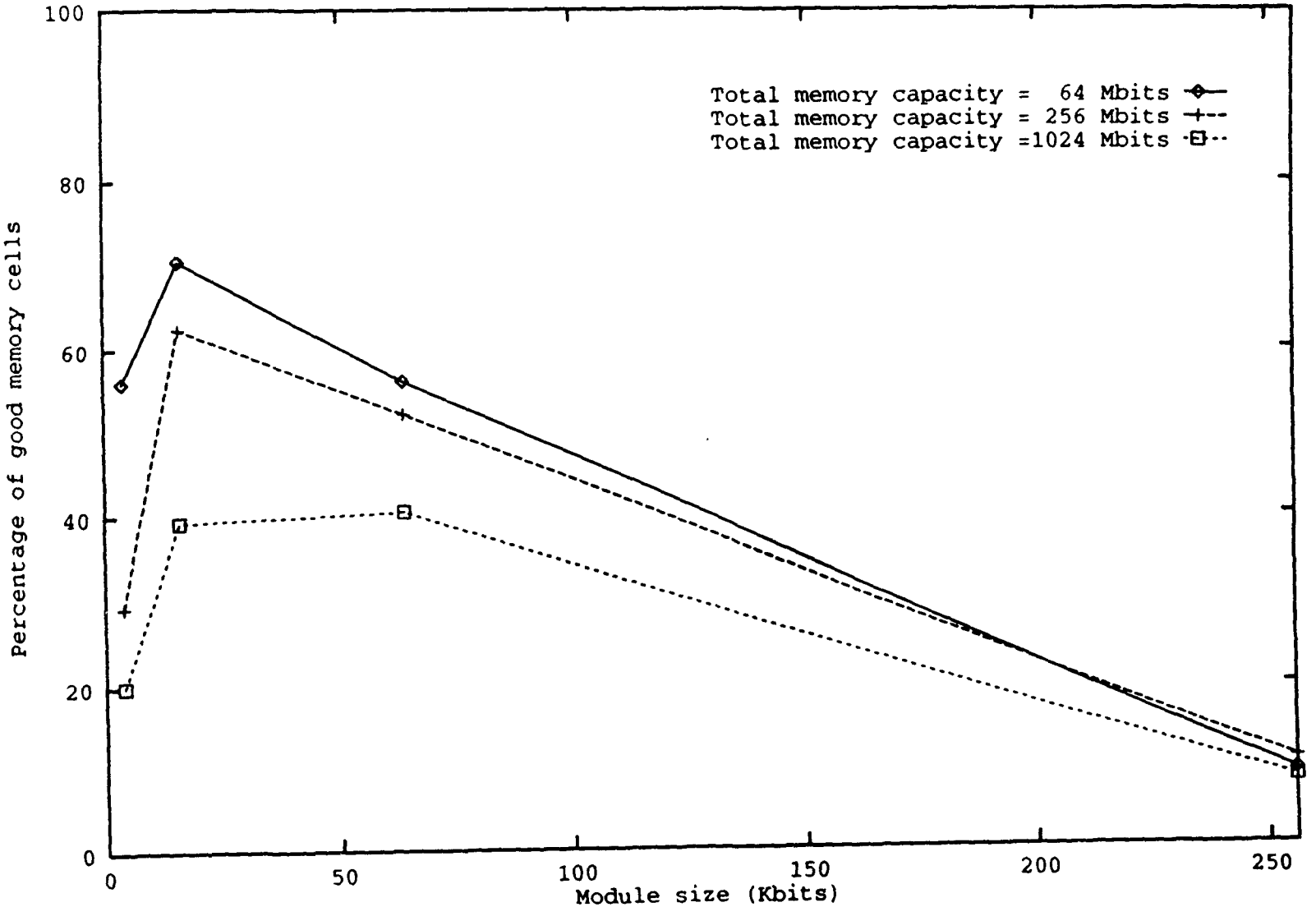

Fig. 5. Yield rate of a WSI memory system (1.0 mu technology, no spares)

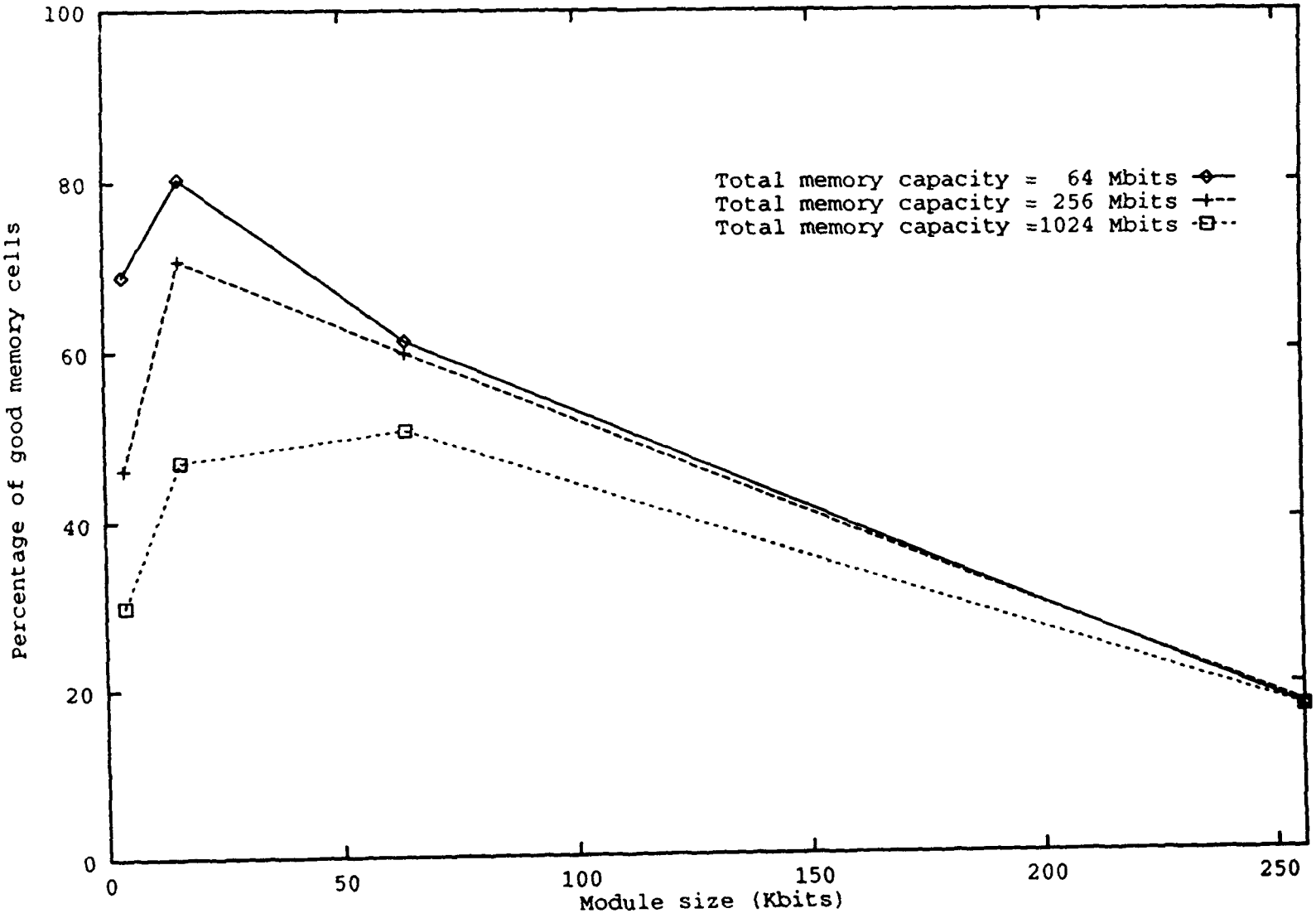

Fig. 6. Yield rate of a WSI memory system (0.6 mu technology, no spares)